1. Preface for Digital Signal Processing: A User's Guide
2. Background, Review, and Reference
    1. Discrete-Time Signals and Systems
    2. Systems in the Time-Domain
    3. Discrete Time Convolution
    4. Review of Linear Algebra
    5. Orthonormal Basis Expansions
    6. Introduction to Fourier Analysis
    7. Continuous Time Fourier Transform (CTFT)
    8. Discrete-Time Fourier Transform (DTFT)
    9. DFT as a Matrix Operation
    10. Sampling theory
        1. Introduction
        2. Proof
        3. Illustrations
        4. Systems view of sampling and reconstruction
        5. Sampling CT Signals: A Frequency Domain Perspective
        6. The DFT: Frequency Domain with a Computer Analysis
        7. Discrete-Time Processing of CT Signals
    11. Z-Transform
        1. Difference Equation
        2. The Z Transform: Definition
        3. Table of Common z-Transforms
        4. Understanding Pole/Zero Plots on the Z-Plane
    12. Random Signals and Processes
        1. Introduction to Random Signals and Processes
        2. Stationary and Nonstationary Random Processes
        3. Random Processes: Mean and Variance
        4. Correlation and Covariance of a Random Signal

        5. Autocorrelation of Random Processes
        6. Crosscorrelation of Random Processes
3. The DFT, FFT, and Practical Spectral Analysis
    1. The Discrete Fourier Transform
        1. DFT Definition and Properties
    2. Spectrum Analysis
        1. Spectrum Analysis Using the Discrete Fourier Transform
        2. Classical Statistical Spectral Estimation
        3. Short Time Fourier Transform
    3. Fast Fourier Transform Algorithms
        1. Overview of Fast Fourier Transform (FFT) Algorithms
        2. Running FFT
        3. Goertzel's Algorithm
        4. Power-of-Two FFTs
            1. Power-of-two FFTs
            2. Radix-2 Algorithms
                1. Decimation-in-time (DIT) Radix-2 FFT
                2. Decimation-in-Frequency (DIF) Radix-2 FFT
                3. Alternate FFT Structures
            3. Radix-4 FFT Algorithms
            4. Split-radix FFT Algorithms
        5. Efficient FFT Algorithm and Programming Tricks
        6. Multidimensional Index Maps
        7. The Prime Factor Algorithm
    4. Fast Convolution
    5. Chirp-z Transform
    6. FFTs of prime length and Rader's conversion
    7. Choosing the Best FFT Algorithm
4. Digital Filter Design

1. Overview of Digital Filter Design
2. FIR Filter Design
    1. Linear Phase Filters
    2. Window Design Method
    3. Frequency Sampling Design Method for FIR filters
    4. Parks-McClellan FIR Filter Design
    5. Lagrange Interpolation
3. IIR Filter Design
    1. Overview of IIR Filter Design
    2. Prototype Analog Filter Design
    3. IIR Digital Filter Design via the Bilinear Transform
    4. Impulse-Invariant Design
    5. Digital-to-Digital Frequency Transformations
    6. Prony's Method
    7. Linear Prediction

5. Digital Filter Structures and Quantization Error Analysis
    1. Filter Structures
        1. Filter Structures
        2. FIR Filter Structures
        3. IIR Filter Structures
        4. State-Variable Representation of Discrete-Time Systems
    2. Fixed-Point Numbers
        1. Fixed-Point Number Representation
        2. Fixed-Point Quantization
    3. Quantization Error Analysis
        1. Finite-Precision Error Analysis
        2. Input Quantization Noise Analysis
        3. Quantization Error in FIR Filters
        4. Data Quantization in IIR Filters
        5. IIR Coefficient Quantization Analysis
    4. Overflow Problems and Solutions

        1. Limit Cycles
        2. Scaling
6. Adaptive Filters and Applications
    1. Introduction to Adaptive Filters
    2. Wiener Filter Algorithm
        1. Discrete-Time, Causal Wiener Filter
        2. Practical Issues in Wiener Filter Implementation
        3. Quadratic Minimization and Gradient Descent
    3. The LMS Adaptive Filter Algorithm
        1. The LMS Adaptive Filter Algorithm
        2. First Order Convergence Analysis of the LMS Algorithm
        3. Second-order Convergence Analysis of the LMS Algorithm and Misadjustment Error
    4. Applications of Adaptive Filters
        1. Applications of Adaptive Filters
        2. Adaptive System Identification
        3. Adaptive Equalization
        4. Adaptive Interference (Noise) Cancellation
        5. Adaptive Echo Cancellation
    5. Other Adaptive Filter Algorithms
        1. Beyond LMS: an overview of other adaptive filter algorithms
        2. Adaptive IIR filters
        3. The Constant-Modulus Algorithm and the Property-Restoral Principle
        4. Complex LMS
        5. Normalized LMS
    6. Summary of Adaptive Filtering Methods
7. Multirate Signal Processing
    1. Overview of Multirate Signal Processing

2. Interpolation, Decimation, and Rate Changing by Integer Fractions
3. Efficient Multirate Filter Structures
4. Filter Design for Multirate Systems
5. Multistage Multirate Systems
6. DFT-Based Filterbanks
7. Quadrature Mirror Filterbanks (QMF)
8. M-Channel Filter Banks

Preface for Digital Signal Processing: A User's Guide

Digital signal processing (DSP) has matured in the past few decades from an obscure research discipline to a large body of practical methods with very broad application. Both practicing engineers and students specializing in signal processing need a clear exposition of the ideas and methods comprising the core signal processing "toolkit" so widely used today.

This text reflects my belief that the skilled practitioner must understand the key ideas underlying the algorithms to select, apply, debug, extend, and innovate most effectively; only with real insight can the engineer make novel use of these methods in the seemingly infinite range of new problems and applications. It also reflects my belief that the needs of the typical student and the practicing engineer have converged in recent years; as the discipline of signal processing has matured, these core topics have become less a subject of active research and more a set of tools applied in the course of other research. The modern student thus has less need for exhaustive coverage of the research literature and detailed derivations and proofs as preparation for their own research on these topics, but greater need for intuition and practical guidance in their most effective use. The majority of students eventually become practicing engineers themselves and benefit from the best preparation for their future careers.

This text both explains the principles of classical signal processing methods and describes how they are used in engineering practice. It is thus much more than a recipe book; it describes the ideas behind the algorithms, gives analyses when they enhance that understanding, and includes derivations that the practitioner may need to extend when applying these methods to new situations. Analyses or derivations that are only of research interest or that do not increase intuitive understanding are left to the references. It is also much more than a theory book; it contains more description of common applications, discussion of actual implementation issues, comments on what really works in the real world, and practical "know-how" than found in the typical academic textbook. The choice of material emphasizes those methods that have found widespread practical use; techniques that have been the subject of intense research but which are

rarely used in practice (for example, RLS adaptive filter algorithms) often receive only limited coverage.

The text assumes a familiarity with basic signal processing concepts such as ideal sampling theory, continuous and discrete Fourier transforms, convolution and filtering. It evolved from a set of notes for a second signal processing course, ECE 451: Digital Signal Processing II, in Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign, aimed at second-semester seniors or first-semester graduate students in signal processing. Over the years, it has been enhanced substantially to include descriptions of common applications, sometimes hard-won knowledge about what actually works and what doesn't, useful tricks, important extensions known to experienced engineers but rarely discussed in academic texts, and other relevant "know-how" to aid the real-world user. This is necessarily an ongoing process, and I continue to expand and refine this component as my own practical knowledge and experience grows. The topics are the core signal processing methods that are used in the majority of signal processing applications; discrete Fourier analysis and FFTs, digital filter design, adaptive filtering, multirate signal processing, and efficient algorithm implementation and finite-precision issues. While many of these topics are covered at an introductory level in a first course, this text aspires to cover all of the methods, both basic and advanced, in these areas which see widespread use in practice. I have also attempted to make the individual modules and sections somewhat self-sufficient, so that those who seek specific information on a single topic can quickly find what they need. Hopefully these aspirations will eventually be achieved; in the meantime, I welcome your comments, corrections, and feedback so that I can continue to improve this text.

As of August 2006, the majority of modules are unedited transcriptions of handwritten notes and may contain typographical errors and insufficient descriptive text for documents unaccompanied by an oral lecture; I hope to have all of the modules in at least presentable shape by the end of the year.

Publication of this text in Connexions would have been impossible without the help of many people. A huge thanks to the various permanent and temporary staff at Connexions is due, in particular to those who converted

the text and equations from my original handwritten notes into CNXML and MathML. My former and current faculty colleagues at the University of Illinois who have taught the second DSP course over the years have had a substantial influence on the evolution of the content, as have the students who have inspired this work and given me feedback. I am very grateful to my teachers, mentors, colleagues, collaborators, and fellow engineers who have taught me the art and practice of signal processing; this work is dedicated to you.

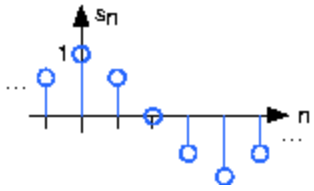
Discrete-Time Signals and Systems
(Blank Abstract)

Mathematically, analog signals are functions having as their independent variables continuous quantities, such as space and time. Discrete-time signals are functions defined on the integers; they are sequences. As with analog signals, we seek ways of decomposing discrete-time signals into simpler components. Because this approach leads to a better understanding of signal structure, we can exploit that structure to represent information (create ways of representing information with signals) and to extract information (retrieve the information thus represented). For symbolic-valued signals, the approach is different: We develop a common representation of all symbolic-valued signals so that we can embody the information they contain in a unified way. From an information representation perspective, the most important issue becomes, for both real-valued and symbolic-valued signals, efficiency: what is the most parsimonious and compact way to represent information so that it can be extracted later.

Real- and Complex-valued Signals

A discrete-time signal is represented symbolically as $s(n)$, where $n = \{\ldots, -1, 0, 1, \ldots\}$.

Cosine



The discrete-time cosine signal is plotted as a stem plot. Can you find the formula for this signal?

We usually draw discrete-time signals as stem plots to emphasize the fact they are functions defined only on the integers. We can delay a discrete-time signal by an integer just as with analog ones. A signal delayed by m samples has the expression $s(n-m)$.

Complex Exponentials

The most important signal is, of course, the **complex exponential sequence**.
Equation:

$$s(n) = e^{i2\pi fn}$$

Note that the frequency variable f is dimensionless and that adding an integer to the frequency of the discrete-time complex exponential has no effect on the signal's value.
Equation:

$$\begin{aligned} e^{i2\pi(f+m)n} &= e^{i2\pi fn}e^{i2\pi mn} \\ &= e^{i2\pi fn} \end{aligned}$$

This derivation follows because the complex exponential evaluated at an integer multiple of 2π equals one. Thus, we need only consider frequency to have a value in some unit-length interval.

Sinusoids

Discrete-time sinusoids have the obvious form $s(n) = A\cos(2\pi fn + \varphi)$. As opposed to analog complex exponentials and sinusoids that can have their frequencies be any real value, frequencies of their discrete-time counterparts yield unique waveforms **only** when f lies in the interval

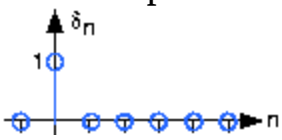
$\left(-\frac{1}{2}, \frac{1}{2}\right]$. This choice of frequency interval is arbitrary; we can also choose the frequency to lie in the interval $[0, 1)$. How to choose a unit-length interval for a sinusoid's frequency will become evident later.

Unit Sample

The second-most important discrete-time signal is the **unit sample**, which is defined to be
Equation:

$$\delta(n) = \begin{cases} 1 & \text{if } n = 0 \\ 0 & \text{otherwise} \end{cases}$$

Unit sample



The unit sample.

Examination of a discrete-time signal's plot, like that of the cosine signal shown in [link], reveals that all signals consist of a sequence of delayed and scaled unit samples. Because the value of a sequence at each integer m is denoted by $s(m)$ and the unit sample delayed to occur at m is written $\delta(n - m)$, we can decompose **any** signal as a sum of unit samples delayed to the appropriate location and scaled by the signal value.
Equation:

$$s(n) = \sum_{m=-\infty}^{\infty} s(m)\delta(n - m)$$

This kind of decomposition is unique to discrete-time signals, and will prove useful subsequently.

Unit Step

The **unit step** in discrete-time is well-defined at the origin, as opposed to the situation with analog signals.
Equation:

$$u(n) = \begin{cases} 1 & \text{if } n \geq 0 \\ 0 & \text{if } n < 0 \end{cases}$$

Symbolic Signals

An interesting aspect of discrete-time signals is that their values do not need to be real numbers. We do have real-valued discrete-time signals like the sinusoid, but we also have signals that denote the sequence of characters typed on the keyboard. Such characters certainly aren't real numbers, and as a collection of possible signal values, they have little mathematical structure other than that they are members of a set. More formally, each element of the **symbolic-valued** signal $s(n)$ takes on one of the values $\{a_1, \ldots, a_K\}$ which comprise the **alphabet** A. This technical terminology does not mean we restrict symbols to being members of the English or Greek alphabet. They could represent keyboard characters, bytes (8-bit quantities), integers that convey daily temperature. Whether controlled by software or not, discrete-time systems are ultimately constructed from digital circuits, which consist **entirely** of analog circuit elements. Furthermore, the transmission and reception of discrete-time signals, like e-mail, is accomplished with analog signals and systems. Understanding how discrete-time and analog signals and systems intertwine is perhaps the main goal of this course.

Discrete-Time Systems

Discrete-time systems can act on discrete-time signals in ways similar to those found in analog signals and systems. Because of the role of software

in discrete-time systems, many more different systems can be envisioned and "constructed" with programs than can be with analog signals. In fact, a special class of analog signals can be converted into discrete-time signals, processed with software, and converted back into an analog signal, all without the incursion of error. For such signals, systems can be easily produced in software, with equivalent analog realizations difficult, if not impossible, to design.

Systems in the Time-Domain

A discrete-time signal $s(n)$ is **delayed** by n_0 samples when we write $s(n - n_0)$, with $n_0 > 0$. Choosing n_0 to be negative advances the signal along the integers. As opposed to analog delays, discrete-time delays can **only** be integer valued. In the frequency domain, delaying a signal corresponds to a linear phase shift of the signal's discrete-time Fourier transform: $s(n - n_0) \leftrightarrow e^{-(i2\pi f n_0)} S\left(e^{i2\pi f}\right)$.

Linear discrete-time systems have the superposition property.
Equation:

Superposition

$$S(a_1 x_1(n) + a_2 x_2(n)) = a_1 S(x_1(n)) + a_2 S(x_2(n))$$

A discrete-time system is called **shift-invariant** (analogous to time-invariant analog systems) if delaying the input delays the corresponding output.
Equation:

Shift-Invariant

$$\text{If } \; S(x(n)) = y(n), \text{ Then } \; S(x(n - n_0)) = y(n - n_0)$$

We use the term shift-invariant to emphasize that delays can only have integer values in discrete-time, while in analog signals, delays can be arbitrarily valued.

We want to concentrate on systems that are both linear and shift-invariant. It will be these that allow us the full power of frequency-domain analysis and implementations. Because we have no physical constraints in "constructing" such systems, we need only a mathematical specification. In analog systems, the differential equation specifies the input-output relationship in the time-domain. The corresponding discrete-time specification is the **difference equation**.
Equation:

The Difference Equation

$$y(n) = a_1 y(n-1) + \ldots + a_p y(n-p) + b_0 x(n) + b_1 x(n-1) + \ldots + b_q x(n-q)$$

Here, the output signal $y(n)$ is related to its **past** values $y(n - l)$, $l = \{1, \ldots, p\}$, and to the current and past values of the input signal $x(n)$. The system's characteristics are determined by the choices for the number of coefficients p and q and the coefficients' values $\{a_1, \ldots, a_p\}$ and $\{b_0, b_1, \ldots, b_q\}$.

Note: There is an asymmetry in the coefficients: where is a_0 ? This coefficient would multiply the $y(n)$ term in the difference equation. We have essentially divided the equation by it, which does not change the input-output relationship. We have thus created the convention that a_0 is always one.

As opposed to differential equations, which only provide an **implicit** description of a system (we must somehow solve the differential equation), difference equations provide an **explicit** way of computing the output for any input. We simply express the difference equation by a program that calculates each output from the previous output values, and the current and previous inputs.

Discrete Time Convolution
Convolution is a concept that extends to all systems that are both linear and time-invariant (LTI). It will become apparent in this discussion that this condition is necessary by demonstrating how linearity and time-invariance give rise to convolution.

Introduction

Convolution, one of the most important concepts in electrical engineering, can be used to determine the output a system produces for a given input signal. It can be shown that a linear time invariant system is completely characterized by its impulse response. The sifting property of the discrete time impulse function tells us that the input signal to a system can be represented as a sum of scaled and shifted unit impulses. Thus, by linearity, it would seem reasonable to compute of the output signal as the sum of scaled and shifted unit impulse responses. That is exactly what the operation of convolution accomplishes. Hence, convolution can be used to determine a linear time invariant system's output from knowledge of the input and the impulse response.

Convolution and Circular Convolution

Convolution

Operation Definition

Discrete time convolution is an operation on two discrete time signals defined by the integral
Equation:

$$(f*g)[n] = \sum_{k=-\infty}^{\infty} f[k]g[n-k]$$

for all signals f, g defined on $\mathbb{Z}$. It is important to note that the operation of convolution is commutative, meaning that
Equation:

$$f*g = g*f$$

for all signals f, g defined on $\mathbb{Z}$. Thus, the convolution operation could have been just as easily stated using the equivalent definition
Equation:

$$(f*g)[n] = \sum_{k=-\infty}^{\infty} f[n-k]g[k]$$

for all signals f, g defined on $\mathbb{Z}$. Convolution has several other important properties not listed here but explained and derived in a later module.

Definition Motivation

The above operation definition has been chosen to be particularly useful in the study of linear time invariant systems. In order to see this, consider a linear time invariant system H with unit impulse response h. Given a system input signal x we would like to compute the system output signal $H(x)$. First, we note that the input can be expressed as the convolution
Equation:

$$x[n] = \sum_{k=-\infty}^{\infty} x[k]\delta[n-k]$$

by the sifting property of the unit impulse function. By linearity
Equation:

$$H(x[n]) = \sum_{k=-\infty}^{\infty} x[k]H(\delta[n-k]).$$

Since $H(\delta[n-k])$ is the shifted unit impulse response $h[n-k]$, this gives the result
Equation:

$$H\left(x\left[n\right]\right) = \sum_{k=-\infty}^{\infty} x\left[k\right] h\left[n-k\right] = \left(x^{*} h\right)\left[n\right].$$

Hence, convolution has been defined such that the output of a linear time invariant system is given by the convolution of the system input with the system unit impulse response.

Graphical Intuition

It is often helpful to be able to visualize the computation of a convolution in terms of graphical processes. Consider the convolution of two functions f, g given by
Equation:

$$\left(f^{*} g\right)\left[n\right] = \sum_{k=-\infty}^{\infty} f\left[k\right] g\left[n-k\right] = \sum_{k=-\infty}^{\infty} f\left[n-k\right] g\left[k\right].$$

The first step in graphically understanding the operation of convolution is to plot each of the functions. Next, one of the functions must be selected, and its plot reflected across the $k = 0$ axis. For each real n, that same function must be shifted left by n. The point-wise product of the two resulting plots is then computed, and then all of the values are summed.

Example:
Recall that the impulse response for a discrete time echoing feedback system with gain a is

Equation:

$$h\left[n\right] = a^{n}u\left[n\right],$$

and consider the response to an input signal that is another exponential
Equation:

$$x\left[n\right] = b^{n}u\left[n\right].$$

We know that the output for this input is given by the convolution of the impulse response with the input signal
Equation:

$$y[n] = x[n]*h[n].$$

We would like to compute this operation by beginning in a way that minimizes the algebraic complexity of the expression. However, in this case, each possible choice is equally simple. Thus, we would like to compute
Equation:

$$y\left[n\right] = \sum_{k=-\infty}^{\infty} a^{k}u\left[k\right]b^{n-k}u\left[n-k\right].$$

The step functions can be used to further simplify this sum. Therefore,
Equation:

$$y[n] = 0$$

for $n < 0$ and
Equation:

$$y\left[n\right] = \sum_{k=0}^{n} \left[ab\right]^{k}$$

for $n \geq 0$. Hence, provided $ab \neq 1$, we have that
Equation:

$$y[n] = \begin{cases} 0 & n < 0 \\ \frac{1-(ab)^{n+1}}{1-(ab)} & n \geq 0 \end{cases}.$$

Circular Convolution

Discrete time circular convolution is an operation on two finite length or periodic discrete time signals defined by the sum
Equation:

$$(f \circledast g)[n] = \sum_{k=0}^{N-1} \widehat{f}[k] \hat{g}[n-k]$$

for all signals f, g defined on $\mathbb{Z}[0, N-1]$ where $\widehat{f}, \hat{g}$ are periodic extensions of f and g. It is important to note that the operation of circular convolution is commutative, meaning that
Equation:

$$f \circledast g = g \circledast f$$

for all signals f, g defined on $\mathbb{Z}[0, N-1]$. Thus, the circular convolution operation could have been just as easily stated using the equivalent definition
Equation:

$$(f \circledast g)[n] = \sum_{k=0}^{N-1} \widehat{f}[n-k] \hat{g}[k]$$

for all signals f, g defined on $\mathbb{Z}[0, N-1]$ where $\widehat{f}, \hat{g}$ are periodic extensions of f and g. Circular convolution has several other important properties not listed here but explained and derived in a later module.

Alternatively, discrete time circular convolution can be expressed as the sum of two summations given by
Equation:

$$(f \circledast g)[n] = \sum_{k=0}^{n} f[k] g[n-k] + \sum_{k=n+1}^{N-1} f[k] g[n-k+N]$$

for all signals f, g defined on $\mathbb{Z}[0, N-1]$.

Meaningful examples of computing discrete time circular convolutions in the time domain would involve complicated algebraic manipulations dealing with the wrap around behavior, which would ultimately be more confusing than helpful. Thus, none will be provided in this section. Of course, example computations in the time domain are easy to program and demonstrate. However, disrete time circular convolutions are more easily computed using frequency domain tools as will be shown in the discrete time Fourier series section.

Definition Motivation

The above operation definition has been chosen to be particularly useful in the study of linear time invariant systems. In order to see this, consider a linear time invariant system H with unit impulse response h. Given a periodic system input signal x we would like to compute the system output signal $H(x)$. First, we note that the input can be expressed as the circular convolution
Equation:

$$x[n] = \sum_{k=0}^{N-1} \widehat{x}[k] \hat{\delta}[n-k]$$

by the sifting property of the unit impulse function. By linearity,
Equation:

$$H\left(x\left[n\right]\right) = \sum_{k=0}^{N-1} \widehat{x}\left[k\right] H\left(\hat{\delta}\left[n-k\right]\right).$$

Since $H(\delta[n-k])$ is the shifted unit impulse response $h[n-k]$, this gives the result

Equation:

$$H\left(x\left[n\right]\right) = \sum_{k=0}^{N-1} \widehat{x}\left[k\right] \widehat{h}\left[n-k\right] = \left(x \circledast h\right)\left[n\right].$$

Hence, circular convolution has been defined such that the output of a linear time invariant system is given by the convolution of the system input with the system unit impulse response.

Graphical Intuition

It is often helpful to be able to visualize the computation of a circular convolution in terms of graphical processes. Consider the circular convolution of two finite length functions f, g given by

Equation:

$$\left(f \circledast g\right)\left[n\right] = \sum_{k=0}^{N-1} \widehat{f}\left[k\right] \hat{g}\left[n-k\right] = \sum_{k=0}^{N-1} \widehat{f}\left[n-k\right] \hat{g}\left[k\right].$$

The first step in graphically understanding the operation of convolution is to plot each of the periodic extensions of the functions. Next, one of the functions must be selected, and its plot reflected across the $k = 0$ axis. For each $n \in \mathbb{Z}[0, N-1]$, that same function must be shifted left by n. The point-wise product of the two resulting plots is then computed, and finally all of these values are summed.

Interactive Element



Interact (when online) with the Mathematica CDF demonstrating Discrete Linear Convolution. To download, right click and save file as .cdf

Convolution Summary

Convolution, one of the most important concepts in electrical engineering, can be used to determine the output signal of a linear time invariant system for a given input signal with knowledge of the system's unit impulse response. The operation of discrete time convolution is defined such that it performs this function for infinite length discrete time signals and systems. The operation of discrete time circular convolution is defined such that it performs this function for finite length and periodic discrete time signals. In each case, the output of the system is the convolution or circular convolution of the input signal with the unit impulse response.

Review of Linear Algebra

Vector spaces are the principal object of study in linear algebra. A vector space is always defined with respect to a field of scalars.

Fields

A field is a set F equipped with two operations, addition and mulitplication, and containing two special members 0 and 1 ($0 \neq 1$), such that for all $\{a, b, c\} \in F$

1. 1. $(a + b) \in F$
   2. $a + b = b + a$
   3. $(a + b) + c = a + (b + c)$
   4. $a + 0 = a$
   5. there exists $-a$ such that $a + -a = 0$

2. 1. $ab \in F$
   2. $ab = ba$
   3. $(ab)c = a\,(bc)$
   4. $a \cdot 1 = a$
   5. there exists a^{-1} such that $aa^{-1} = 1$

3. $a\,(b + c) = ab + ac$

More concisely

1. F is an abelian group under addition
2. F is an abelian group under multiplication
3. multiplication distributes over addition

Examples

$\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$

Vector Spaces

Let F be a field, and V a set. We say V **is a vector space over** F if there exist two operations, defined for all $a \in F$, $\boldsymbol{u} \in V$ and $\boldsymbol{v} \in V$:

- vector addition: $(\boldsymbol{u}, \boldsymbol{v}) \to (\boldsymbol{u} + \boldsymbol{v}) \in V$
- scalar multiplication: $(a, \boldsymbol{v}) \to a\boldsymbol{v} \in V$

and if there exists an element denoted $\quad \in V$, such that the following hold for all $a \in F$, $b \in F$, and $\boldsymbol{u} \in V$, $\boldsymbol{v} \in V$, and $\boldsymbol{w} \in V$

1. 1. $\boldsymbol{u} + (\boldsymbol{v} + \boldsymbol{w}) = (\boldsymbol{u} + \boldsymbol{v}) + \boldsymbol{w}$
   2. $\boldsymbol{u} + \boldsymbol{v} = \boldsymbol{v} + \boldsymbol{u}$
   3. $\boldsymbol{u} + \quad = \boldsymbol{u}$
   4. there exists $-\boldsymbol{u}$ such that $\boldsymbol{u} + -\boldsymbol{u} =$

2. 1. $a\,(\boldsymbol{u} + \boldsymbol{v}) = a\boldsymbol{u} + a\boldsymbol{v}$
   2. $(a + b)\boldsymbol{u} = a\boldsymbol{u} + b\boldsymbol{u}$
   3. $(ab)\boldsymbol{u} = a\,(b\boldsymbol{u})$
   4. $1 \cdot \boldsymbol{u} = \boldsymbol{u}$

More concisely,

1. V is an abelian group under plus
2. Natural properties of scalar multiplication

Examples

- ^{N} is a vector space over $\mathbb{R}$
- ^{N} is a vector space over $\mathbb{C}$
- ^{N} is a vector space over $\mathbb{R}$
- ^{N} is **not** a vector space over $\mathbb{C}$

The elements of V are called **vectors**.

Euclidean Space

Throughout this course we will think of a signal as a vector

$$\boldsymbol{x} = \begin{matrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{matrix} = \begin{pmatrix} x_1 & x_2 & \dots & x_N \end{pmatrix}^T$$

The samples $\{x_i\}$ could be samples from a finite duration, continuous time signal, for example.

A signal will belong to one of two vector spaces:

Real Euclidean space

$\boldsymbol{x} \in \quad^N$ (over $\mathbb{R}$)

Complex Euclidean space

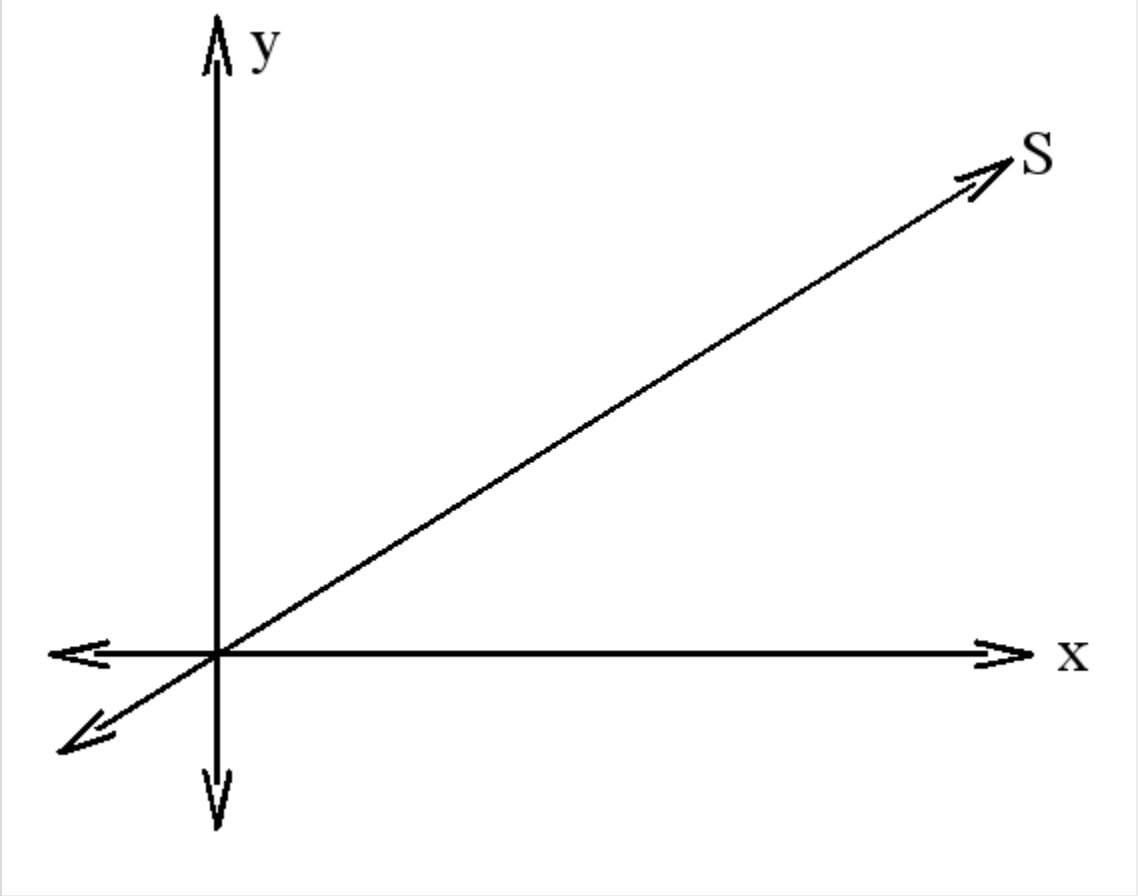
$\boldsymbol{x} \in \quad^N$ (over $\mathbb{C}$)

Subspaces

Let V be a vector space over F.

A subset $S \subseteq V$ is called a **subspace** of V if S is a vector space over F in its own right.

Example:
$V = \quad^2$, $F = \quad$, $S =$ any line though the origin.



S is any line through the origin.

Are there other subspaces?

$S \subseteq V$ is a subspace if and only if for all $a \in F$ and $b \in F$ and for all $\boldsymbol{s} \in S$ and $\boldsymbol{t} \in S$, $(a\boldsymbol{s} + b\boldsymbol{t}) \in S$

Linear Independence

Let $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_k \in V$.

We say that these vectors are **linearly dependent** if there exist scalars $a_1, \ldots, a_k \in F$ such that

Equation:

$$\sum_{i=1}^{k} a_i u_i =$$

and at least one $a_i \neq 0$.

If [link] only holds for the case $a_1 = \ldots = a_k = 0$, we say that the vectors are **linearly independent**.

Example:

$$1 \begin{vmatrix} 1 \\ -1 \\ 2 \end{vmatrix} - 2 \begin{vmatrix} -2 \\ 3 \\ 0 \end{vmatrix} + 1 \begin{vmatrix} -5 \\ 7 \\ -2 \end{vmatrix} =$$

so these vectors are linearly dependent in ^{3}.

Spanning Sets

Consider the subset $S = \{v_1, v_2, \ldots, v_k\}$. Define the **span** of S

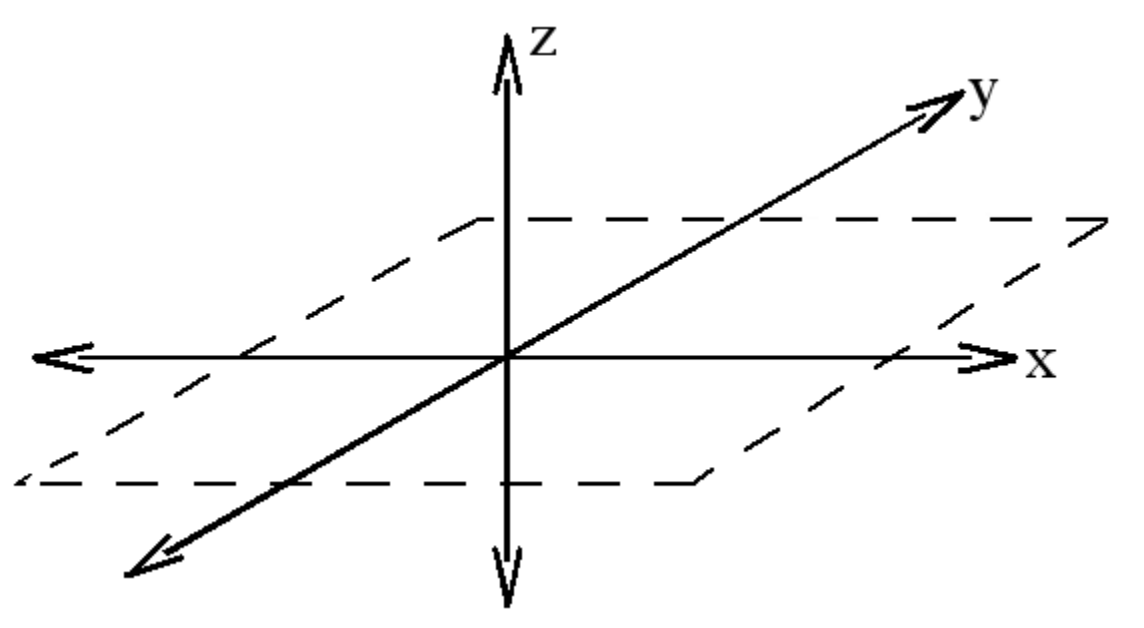
$$< S > \equiv \text{span}\,(S) \equiv \sum_{i=1}^{k} a_i v_i \;\; a_i \in F$$

Fact: $< S >$ is a subspace of V.

Example:

$V = \quad ^3, F = \quad , S = \{v_1, v_2\}, v_1 = \begin{matrix} 1 \\ 0 \\ 0 \end{matrix}, v_2 = \begin{matrix} 0 \\ 1 \\ 0 \end{matrix} \Rightarrow$

$< S >=$ xy-plane.



$< S >$ is the xy-plane.

Aside

If S is infinite, the notions of linear independence and span are easily generalized:

We say S is linearly independent if, for every finite collection $u_1, \ldots, u_k \in S$, (k arbitrary) we have

$$\sum_{i=1}^{k} a_i u_i = \quad \Rightarrow \forall i : (a_i = 0)$$

The span of S is

$$< S >= \sum_{i=1}^{k} a_i u_i \;\; a_i \in F \;\wedge\; u_i \in S \;\wedge\; (k < \infty)$$

Note: In both definitions, we only consider **finite** sums.

Bases

A set $B \subseteq V$ is called a **basis** for V over F if and only if

1. B is linearly independent
2. $< B >= V$

Bases are of fundamental importance in signal processing. They allow us to decompose a signal into building blocks (basis vectors) that are often more easily understood.

Example:
V = (real or complex) Euclidean space, $\mathbb{R}^N$ or $\mathbb{C}^N$.

$$B = \{e_1, \ldots, e_N\} \equiv \text{standard basis}$$

$$e_i = \begin{matrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{matrix}$$

where the 1 is in the ith position.

Example:
$V = \quad ^N$ over $\mathbb{C}$.

$$B = \{u_1, \ldots, u_N\}$$

which is the DFT basis.

$$u_k = \begin{matrix} 1 \\ e^{-\ i2\pi\frac{k}{N}} \\ \vdots \\ e^{-\ i2\pi\frac{k}{N}(N-1)} \end{matrix}$$

where $i = \sqrt{-1}$.

Key Fact

If B is a basis for V, then every $\boldsymbol{v} \in V$ can be written uniquely (up to order of terms) in the form

$$\boldsymbol{v} = \sum_{i\ \ 1}^{N} a_i v_i$$

where $a_i \in F$ and $v_i \in B$.

Other Facts

- If S is a linearly independent set, then S can be extended to a basis.
- If $< S >= V$, then S contains a basis.

Dimension

Let V be a vector space with basis B. The dimension of V, denoted $\dim(V)$, is the cardinality of B.

Every vector space has a basis.

Every basis for a vector space has the same cardinality.

$\Rightarrow \dim(V)$ is **well-defined**.

If $\dim(V) < \infty$, we say V is **finite dimensional**.

Examples

vector space	field of scalars	dimension
N		
N		
N		

Every subspace is a vector space, and therefore has its own dimension.

Example:
Suppose $(S = \{u_1, \ldots, u_k\}) \subseteq V$ is a linearly independent set. Then

$$\dim(< S >) =$$

Facts

- If S is a subspace of V, then $\dim(S) \leq \dim(V)$.
- If $\dim(S) = \dim(V) < \infty$, then $S = V$.

Direct Sums

Let V be a vector space, and let $S \subseteq V$ and $T \subseteq V$ be subspaces.

We say V is the **direct sum** of S and T, written $V = S \oplus T$, if and only if for every $\boldsymbol{v} \in V$, there exist unique $\boldsymbol{s} \in S$ and $\boldsymbol{t} \in T$ such that $\boldsymbol{v} = \boldsymbol{s} + \boldsymbol{t}$.

If $V = S \oplus T$, then T is called a **complement** of S.

Example:

$$V = C' = \{f : \quad \to \quad | f \text{is continuous}\}$$

$$S = \text{even funcitons in} C'$$

$$T = \text{odd funcitons in} C'$$

$$f(t) = \frac{1}{2}\left(f(t) + f(-t)\right) + \frac{1}{2}\left(f(t) - f(-t)\right)$$

If $f = g + h = g' + h'$, $g \in S$ and $g' \in S$, $h \in T$ and $h' \in T$, then $g - g' = h' - h$ is odd and even, which implies $g = g'$ and $h = h'$.

Facts

1. Every subspace has a complement
2. $V = S \oplus T$ if and only if
    1. $S \cap T = \{\ \}$
    2. $< S, T >= V$

3. If $V = S \oplus T$, and $\dim(V) < \infty$, then
   $\dim(V) = \dim(S) + \dim(T)$

Proofs

Invoke a basis.

Norms

Let V be a vector space over F. A norm is a mapping $V \to F$, denoted by $\| \quad \|$, such that forall $\boldsymbol{u} \in V$, $\boldsymbol{v} \in V$, and $\lambda \in F$

1. $\| \boldsymbol{u} \| > 0$ if $\boldsymbol{u} \neq$
2. $\| \lambda \boldsymbol{u} \| = |\lambda| \| \boldsymbol{u} \|$
3. $\| \boldsymbol{u} + \boldsymbol{v} \| \leq \| \boldsymbol{u} \| + \| \boldsymbol{v} \|$

Examples

Euclidean norms:

$\boldsymbol{x} \in \quad^N$:

$$\| \boldsymbol{x} \| = \sum_{i \quad 1}^{N} {x_i}^2 \quad^{\frac{1}{2}}$$

$\boldsymbol{x} \in \quad^N$:

$$\| \boldsymbol{x} \| = \sum_{i \quad 1}^{N} \left(|x_i| \right)^2 \quad^{\frac{1}{2}}$$

Induced Metric

Every norm induces a metric on V

$$d(\boldsymbol{u}, \boldsymbol{v}) \equiv \| \boldsymbol{u} - \boldsymbol{v} \|$$

which leads to a notion of "distance" between vectors.

Inner products

Let V be a vector space over F, $F =$ or . An inner product is a mapping $V \times V \to F$, denoted $\langle \cdot, \cdot \rangle$, such that

1. $\langle \boldsymbol{v}, \boldsymbol{v} \rangle \geq 0$, and $\langle \boldsymbol{v}, \boldsymbol{v} \rangle = 0 \Leftrightarrow \boldsymbol{v} = 0$
2. $\langle \boldsymbol{u}, \boldsymbol{v} \rangle = \langle \boldsymbol{v}, \boldsymbol{u} \rangle$
3. $\langle a\boldsymbol{u} + b\boldsymbol{v}, \boldsymbol{w} \rangle = a \langle (\boldsymbol{u}, \boldsymbol{w}) \rangle + b \langle (\boldsymbol{v}, \boldsymbol{w}) \rangle$

Examples

^{N} over $\mathbb{R}$:

$$\langle \boldsymbol{x}, \boldsymbol{y} \rangle = \boldsymbol{x}^T \boldsymbol{y} = \sum_{i \ 1}^{N} x_i y_i$$

^{N} over $\mathbb{C}$:

$$\langle \boldsymbol{x}, \boldsymbol{y} \rangle = \boldsymbol{x}^{\mathrm{H}} \boldsymbol{y} = \sum_{i \ 1}^{N} x_i y_i$$

If $\boldsymbol{x} = (x_1 \ldots x_N)^T \in$, then

$$\boldsymbol{x}^{\mathrm{H}} \equiv \begin{matrix} x_1 \\ \vdots \\ x_N \end{matrix}^{T}$$

is called the "Hermitian," or "conjugate transpose" of $\boldsymbol{x}$.

Triangle Inequality

If we define $\| \boldsymbol{u} \| = \langle \boldsymbol{u}, \boldsymbol{u} \rangle$, then

$$\| \boldsymbol{u} + \boldsymbol{v} \| \leq \| \boldsymbol{u} \| + \| \boldsymbol{v} \|$$

Hence, every inner product induces a norm.

Cauchy-Schwarz Inequality

For all $\boldsymbol{u} \in V$, $\boldsymbol{v} \in V$,

$$|\langle \boldsymbol{u}, \boldsymbol{v} \rangle| \leq \| \boldsymbol{u} \| \ \ \| \boldsymbol{v} \|$$

In inner product spaces, we have a notion of the angle between two vectors:

$$\angle(\boldsymbol{u}, \boldsymbol{v}) = \arccos \ \ \frac{\langle \boldsymbol{u}, \boldsymbol{v} \rangle}{\| \boldsymbol{u} \| \ \ \| \boldsymbol{v} \|} \qquad \in [0, 2\pi)$$

Orthogonality

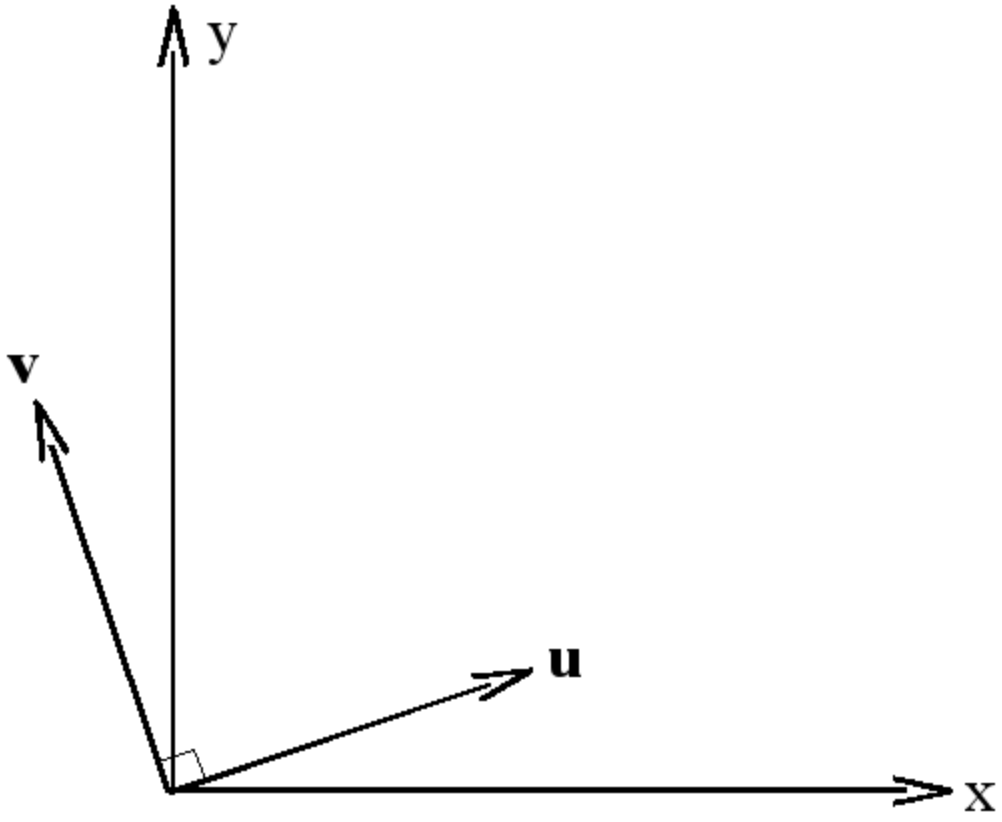
$\boldsymbol{u}$ and $\boldsymbol{v}$ are **orthogonal** if

$$\langle \boldsymbol{u}, \boldsymbol{v} \rangle = 0$$

Notation: $\boldsymbol{u} \perp \boldsymbol{v}$.

If in addition $\| \boldsymbol{u} \| = \| \boldsymbol{v} \| = 1$, we say $\boldsymbol{u}$ and $\boldsymbol{v}$ are **orthonormal**.

In an orthogonal (orthonormal) **set**, each pair of vectors is orthogonal (orthonormal).



Orthogonal vectors in ^{2}.

Orthonormal Bases

An Orthonormal basis is a basis $\{v_i\}$ such that

$$\langle v_i, v_i \rangle = \delta_{ij} = \begin{matrix} 1 \text{ if } i = j \\ 0 \text{ if } i \neq j \end{matrix}$$

Example:
The standard basis for ^{N} or ^{N}

Example:
The normalized DFT basis

$$u_k = \frac{1}{\sqrt{N}} \begin{matrix} 1 \\ e^{-\ i2\pi\frac{k}{N}} \\ \vdots \\ e^{-\ i2\pi\frac{k}{N}(N-1)} \end{matrix}$$

Expansion Coefficients

If the representation of $\boldsymbol{v}$ with respect to $\{v_i\}$ is

$$\boldsymbol{v} = \sum_i a_i v_i$$

then

$$a_i = \langle v_i, \boldsymbol{v} \rangle$$

Gram-Schmidt

Every inner product space has an orthonormal basis. Any (countable) basis can be made orthogonal by the Gram-Schmidt orthogonalization process.

Orthogonal Compliments

Let $S \subseteq V$ be a subspace. The **orthogonal compliment** S is

$$S^{\perp} = \{\boldsymbol{u} | \, \boldsymbol{u} \in V \ \wedge \ (\langle \boldsymbol{u}, \boldsymbol{v} \rangle = 0) \ \wedge \ \forall \boldsymbol{v} : (\boldsymbol{v} \in S)\}$$

$S^{\perp}$ is easily seen to be a subspace.

If $\dim(v) < \infty$, then $V = S \oplus S^{\perp}$.

Note: If $\dim(v) = \infty$, then in order to have $V = S \oplus S^{\perp}$ we require V to be a **Hilbert Space**.

Linear Transformations

Loosely speaking, a linear transformation is a mapping from one vector space to another that **preserves** vector space operations.

More precisely, let V, W be vector spaces over the same field F. A **linear transformation** is a mapping $T : V \to W$ such that

$$T(a\boldsymbol{u} + b\boldsymbol{v}) = aT(\boldsymbol{u}) + bT(\boldsymbol{v})$$

for all $a \in F$, $b \in F$ and $\boldsymbol{u} \in V$, $\boldsymbol{v} \in V$.

In this class we will be concerned with linear transformations between (real or complex) **Euclidean spaces**, or subspaces thereof.

Image

$$\text{image}\,(T) = \{\boldsymbol{w} | \, \boldsymbol{w} \in W \ \wedge \ T(\boldsymbol{v}) = \boldsymbol{w} \text{for some} \boldsymbol{v}\}$$

Nullspace

Also known as the kernel:

$$\ker(T) = \{\boldsymbol{v} | \, \boldsymbol{v} \in V \ \wedge \ (T(\boldsymbol{v}) = \ \ )\}$$

Both the image and the nullspace are easily seen to be subspaces.

Rank

$$\operatorname{rank}(T) = \dim(\text{image}\ (T))$$

Nullity

$$\operatorname{null}(T) = \dim(\ker(T))$$

Rank plus nullity theorem

$$\operatorname{rank}(T) + \operatorname{null}(T) = \dim(V)$$

Matrices

Every linear transformation T has a **matrix representation**. If $T:\ ^N \to\ ^M,\quad =\quad$ or $\quad$, then T is represented by an $M \times N$ matrix

$$A = \begin{matrix} a_{11} & \dots & a_{1N} \\ \vdots & \ddots & \vdots \\ a_{M1} & \dots & a_{MN} \end{matrix}$$

where $(a_{1i} \dots a_{Mi})^T = T(e_i)$ and $e_i = (0 \dots 1 \dots 0)^T$ is the ith **standard basis** vector.

Note:A linear transformation can be represented with respect to any bases of ^{N} and ^{M}, leading to a different A. We will always represent a linear transformation using the standard bases.

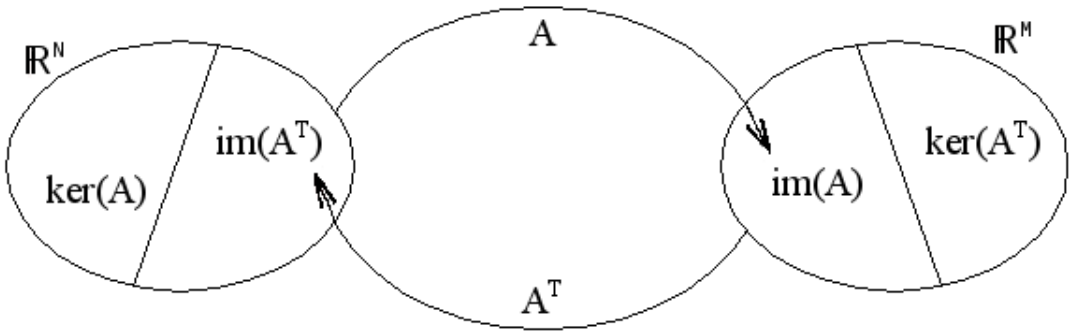
Column span

$$\text{colspan}\,(A) =< A >=\text{image}\,(A)$$

Duality

If $A:\ ^{N}\rightarrow\ ^{M}$, then

$$\ker^{\perp}(A) =\text{image}\ \ A^{T}$$



If $A:\ ^{N}\rightarrow\ ^{M}$, then

$$\ker^{\perp}(A) =\text{image}\ \ A^{\mathrm{H}}$$

Inverses

The linear transformation/matrix A is **invertible** if and only if there exists a matrix B such that $AB = BA = I$ (identity).

Only **square** matrices can be invertible.

Let $A:\ ^{N}\rightarrow\ ^{N}$ be linear, $=$ or . The following are equivalent:

1. A is invertible (nonsingular)
2. $\text{rank}\,(A) = N$
3. $\text{null}\,(A) = 0$

4. $\det A \neq 0$
5. The columns of A form a basis.

If $A^{-1} = A^T$ (or A^{H} in the complex case), we say A is **orthogonal** (or **unitary**).

Orthonormal Basis Expansions
The module looks at decomposing signals through orthonormal basis expansion to provide an alternative representation. The module presents many examples of solving these problems and looks at them in several spaces and dimensions.

Main Idea

When working with signals many times it is helpful to break up a signal into smaller, more manageable parts. Hopefully by now you have been exposed to the concept of eigenvectors and there use in decomposing a signal into one of its possible basis. By doing this we are able to simplify our calculations of signals and systems through eigenfunctions of LTI systems.

Now we would like to look at an alternative way to represent signals, through the use of **orthonormal basis**. We can think of orthonormal basis as a set of building blocks we use to construct functions. We will build up the signal/vector as a weighted sum of basis elements.

Example:
The complex sinusoids $\frac{1}{\sqrt{T}}e^{i\omega_0 nt}$ for all $-\infty < n < \infty$ form an orthonormal basis for $L^2([0,T])$.
In our Fourier series equation, $f(t) = \sum_{n=-\infty}^{\infty} c_n e^{i\omega_0 nt}$, the $\{c_n\}$ are just another representation of $f(t)$.

Note: For signals/vectors in a Hilbert Space, the expansion coefficients are easy to find.

Alternate Representation

Recall our definition of a **basis**: A set of vectors $\{b_i\}$ in a vector space S is a basis if

1. The b_i are linearly independent.
2. The b_i span S. That is, we can find $\{\alpha_i\}$, where $\alpha_i \in \mathbb{C}$ (scalars) such that
   Equation:

$$\forall x, x \in S : \left(\boldsymbol{x} = \sum_i \alpha_i b_i \right)$$

   where $\boldsymbol{x}$ is a vector in S, α is a scalar in $\mathbb{C}$, and $\boldsymbol{b}$ is a vector in S.

Condition 2 in the above definition says we can **decompose** any vector in terms of the $\{b_i\}$. Condition 1 ensures that the decomposition is **unique** (think about this at home).

Note: The $\{\alpha_i\}$ provide an alternate representation of $\boldsymbol{x}$.

Example:
Let us look at simple example in $\mathbb{R}^2$, where we have the following vector:

$$\boldsymbol{x} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

Standard Basis: $\{e_0, e_1\} = \left\{ (10)^T, (01)^T \right\}$

$$\boldsymbol{x} = e_0 + 2e_1$$

Alternate Basis: $\{h_0, h_1\} = \left\{ (11)^T, (1{-}1)^T \right\}$

$$\boldsymbol{x} = \frac{3}{2}h_0 + \frac{-1}{2}h_1$$

In general, given a basis $\{b_0, b_1\}$ and a vector $\boldsymbol{x} \in \mathbb{R}^2$, how do we find the α_0 and α_1 such that
Equation:

$$\boldsymbol{x} = \alpha_0 b_0 + \alpha_1 b_1$$

Finding the Coefficients

Now let us address the question posed above about finding α_i's in general for $\mathbb{R}^2$. We start by rewriting [link] so that we can stack our b_i's as columns in a 2×2 matrix.
Equation:

$$(\boldsymbol{x}) = \alpha_0 (b_0) + \alpha_1 (b_1)$$

Equation:

$$(\boldsymbol{x}) = \begin{pmatrix} \vdots & \vdots \\ b_0 & b_1 \\ \vdots & \vdots \end{pmatrix} \begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix}$$

Example:
Here is a simple example, which shows a little more detail about the above equations.
Equation:

$$\begin{aligned}\begin{pmatrix} x[0] \\ x[1] \end{pmatrix} &= \alpha_0 \begin{pmatrix} b_0[0] \\ b_0[1] \end{pmatrix} + \alpha_1 \begin{pmatrix} b_1[0] \\ b_1[1] \end{pmatrix} \\ &= \begin{pmatrix} \alpha_0 b_0[0] + \alpha_1 b_1[0] \\ \alpha_0 b_0[1] + \alpha_1 b_1[1] \end{pmatrix}\end{aligned}$$

Equation:

$$\begin{pmatrix} x[0] \\ x[1] \end{pmatrix} = \begin{pmatrix} b_0[0] & b_1[0] \\ b_0[1] & b_1[1] \end{pmatrix} \begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix}$$

Simplifying our Equation

To make notation simpler, we define the following two items from the above equations:

- **Basis Matrix**:

$$B = \begin{pmatrix} \vdots & \vdots \\ b_0 & b_1 \\ \vdots & \vdots \end{pmatrix}$$

- **Coefficient Vector**:

$$\boldsymbol{\alpha} = \begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix}$$

This gives us the following, concise equation:
Equation:

$$\boldsymbol{x} = B\boldsymbol{\alpha}$$

which is equivalent to $\boldsymbol{x} = \sum_{i=0}^{1} \alpha_i b_i$.

Example:
Given a standard basis, $\left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}$, then we have the following basis matrix:

$$B = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

To get the α_i's, we solve for the coefficient vector in [link]
Equation:

$$\boldsymbol{\alpha} = B^{-1}\boldsymbol{x}$$

Where B^{-1} is the inverse matrix of B.

Examples

Example:
Let us look at the standard basis first and try to calculate $\boldsymbol{\alpha}$ from it.

$$B = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = I$$

Where I is the **identity matrix**. In order to solve for $\boldsymbol{\alpha}$ let us find the inverse of B first (which is obviously very trivial in this case):

$$B^{-1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Therefore we get,

$$\boldsymbol{\alpha} = B^{-1}\boldsymbol{x} = \boldsymbol{x}$$

Example:
Let us look at a ever-so-slightly more complicated basis of $\left\{\begin{pmatrix}1\\1\end{pmatrix}, \begin{pmatrix}1\\-1\end{pmatrix}\right\} = \{h_0, h_1\}$ Then our basis matrix and inverse basis matrix becomes:

$$B = \begin{pmatrix}1 & 1\\1 & -1\end{pmatrix}$$

$$B^{-1} = \begin{pmatrix}\frac{1}{2} & \frac{1}{2}\\\frac{1}{2} & \frac{-1}{2}\end{pmatrix}$$

and for this example it is given that

$$\boldsymbol{x} = \begin{pmatrix}3\\2\end{pmatrix}$$

Now we solve for $\boldsymbol{\alpha}$

$$\boldsymbol{\alpha} = B^{-1}\boldsymbol{x} = \begin{pmatrix}\frac{1}{2} & \frac{1}{2}\\\frac{1}{2} & \frac{-1}{2}\end{pmatrix}\begin{pmatrix}3\\2\end{pmatrix} = \begin{pmatrix}2.5\\0.5\end{pmatrix}$$

and we get

$$\boldsymbol{x} = 2.5h_0 + 0.5h_1$$

Exercise:

Problem:

Now we are given the following basis matrix and $\boldsymbol{x}$:

$$\{b_0, b_1\} = \left\{ \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \begin{pmatrix} 3 \\ 0 \end{pmatrix} \right\}$$

$$\boldsymbol{x} = \begin{pmatrix} 3 \\ 2 \end{pmatrix}$$

For this problem, make a sketch of the bases and then represent $\boldsymbol{x}$ in terms of b_0 and b_1.

Solution:

In order to represent $\boldsymbol{x}$ in terms of b_0 and b_1 we will follow the same steps we used in the above example.

$$B = \begin{pmatrix} 1 & 2 \\ 3 & 0 \end{pmatrix}$$

$$B^{-1} = \begin{pmatrix} 0 & \frac{1}{2} \\ \frac{1}{3} & \frac{-1}{6} \end{pmatrix}$$

$$\boldsymbol{\alpha} = B^{-1}\boldsymbol{x} = \begin{pmatrix} 1 \\ \frac{2}{3} \end{pmatrix}$$

And now we can write $\boldsymbol{x}$ in terms of b_0 and b_1.

$$\boldsymbol{x} = b_0 + \frac{2}{3} b_1$$

And we can easily substitute in our known values of b_0 and b_1 to verify our results.

Note: A change of basis simply looks at $\boldsymbol{x}$ from a "different perspective." B^{-1} **transforms** $\boldsymbol{x}$ from the standard basis to our new basis, $\{b_0, b_1\}$. Notice that this is a totally mechanical procedure.

Extending the Dimension and Space

We can also extend all these ideas past just $\mathbb{R}^2$ and look at them in $\mathbb{R}^n$ and $\mathbb{C}^n$. This procedure extends naturally to higher (> 2) dimensions. Given a basis $\{b_0, b_1, \ldots, b_{n-1}\}$ for $\mathbb{R}^n$, we want to find $\{\alpha_0, \alpha_1, \ldots, \alpha_{n-1}\}$ such that

Equation:

$$\boldsymbol{x} = \alpha_0 b_0 + \alpha_1 b_1 + \ldots + \alpha_{n-1} b_{n-1}$$

Again, we will set up a basis matrix

$$B = \begin{pmatrix} b_0 & b_1 & b_2 & \ldots & b_{n-1} \end{pmatrix}$$

where the columns equal the basis vectors and it will always be an n×n matrix (although the above matrix does not appear to be square since we left terms in vector notation). We can then proceed to rewrite [link]

$$\boldsymbol{x} = \begin{pmatrix} b_0 & b_1 & \ldots & b_{n-1} \end{pmatrix} \begin{pmatrix} \alpha_0 \\ \vdots \\ \alpha_{n-1} \end{pmatrix} = B\boldsymbol{\alpha}$$

and

$$\boldsymbol{\alpha} = B^{-1}\boldsymbol{x}$$

Introduction to Fourier Analysis
Lists the four Fourier transforms and when to use them.

Fourier's Daring Leap

Fourier postulated around 1807 that any periodic signal (equivalently finite length signal) can be built up as an infinite linear combination of harmonic sinusoidal waves.

i.e. Given the collection
Equation:

$$B = \left\{ e^{j\frac{2\pi}{T}nt} \right\}_{n=-\infty}^{\infty}$$

any
Equation:

$$f(t) \in L^2[0, T)$$

can be approximated arbitrarily closely by
Equation:

$$f(t) = \sum_{n=-\infty}^{\infty} C_n \, e^{j\frac{2\pi}{T}nt}.$$

Now, The issue of exact convergence did bring Fourier much criticism from the French Academy of Science (Laplace, Lagrange, Monge and LaCroix comprised the review committee) for several years after its presentation on 1807. It was not resolved for also a century, and its resolution is interesting and important to understand from a practical viewpoint. See more in the section on **Gibbs Phenomena**.

Fourier analysis is fundamental to understanding the behavior of signals and systems. This is a result of the fact that sinusoids are Eigenfunctions of linear, time-invariant (LTI) systems. This is to say that if we pass any particular sinusoid through a LTI system, we get a scaled version of that same sinusoid on the output. Then, since Fourier analysis allows us to redefine the signals in terms of sinusoids, all we need to do is determine how any given system effects all possible sinusoids (its transfer function) and we have a complete understanding of the system. Furthermore, since we are able to define the passage of sinusoids through a system as multiplication of that sinusoid by the transfer function at the same frequency, we can convert the passage of any signal through a system from convolution (in time) to multiplication (in frequency). These ideas are what give Fourier analysis its power.

Now, after hopefully having sold you on the value of this method of analysis, we must examine exactly what we mean by Fourier analysis. The four Fourier transforms that comprise this analysis are the Fourier Series, Continuous-Time Fourier Transform, Discrete-Time Fourier Transform and Discrete Fourier Transform. For this document, we will view the Laplace Transform and Z-Transform as simply extensions of the CTFT and DTFT respectively. All of these transforms act essentially the same way, by converting a signal in time to an equivalent signal in frequency (sinusoids). However, depending on the nature of a specific signal i.e. whether it is finite- or infinite-length and whether it is discrete- or continuous-time) there is an appropriate transform to convert the signal into the frequency domain. Below is a table of the four Fourier transforms and when each is appropriate. It also includes the relevant convolution for the specified space.

Transform	Time Domain	Frequency Domain	Convolution

Transform	Time Domain	Frequency Domain	Convolution
Continuous-Time Fourier Series	$L^2([0,T))$	$l^2(\ \ )$	Continuous-Time Circular
Continuous-Time Fourier Transform	$L^2(\ \ )$	$L^2(\ \ )$	Continuous-Time Linear
Discrete-Time Fourier Transform	$l^2(\ \ )$	$L^2([0,2\pi))$	Discrete-Time Linear
Discrete Fourier Transform	$l^2([0,N-1])$	$l^2([0,N-1])$	Discrete-Time Circular

Table of Fourier Representations

Continuous Time Fourier Transform (CTFT)
Details the Continuous-Time Fourier Transform.

Introduction

In this module, we will derive an expansion for any arbitrary continuous-time function, and in doing so, derive the **Continuous Time Fourier Transform** (CTFT).

Since complex exponentials are eigenfunctions of linear time-invariant (LTI) systems, calculating the output of an LTI system $\mathscr{H}$ given e^{st} as an input amounts to simple multiplication, where $H(s) \in \mathbb{C}$ is the eigenvalue corresponding to s. As shown in the figure, a simple exponential input would yield the output
Equation:

$$y(t) = H(s)e^{st}$$

Using this and the fact that $\mathscr{H}$ is linear, calculating $y(t)$ for combinations of complex exponentials is also straightforward.

$$c_1 e^{s_1 t} + c_2 e^{s_2 t} \rightarrow c_1 H(s_1) e^{s_1 t} + c_2 H(s_2) e^{s_2 t}$$

$$\sum_n c_n e^{s_n t} \rightarrow \sum_n c_n H(s_n) e^{s_n t}$$

The action of H on an input such as those in the two equations above is easy to explain. $\mathscr{H}$ **independently scales** each exponential component $e^{s_n t}$ by a different complex number $H(s_n) \in \mathbb{C}$. As such, if we can write a function $f(t)$ as a combination of complex exponentials it allows us to easily calculate the output of a system.

Now, we will look to use the power of complex exponentials to see how we may represent arbitrary signals in terms of a set of simpler functions by superposition of a number of complex exponentials. Below we will present the **Continuous-Time Fourier Transform** (CTFT), commonly referred to

as just the Fourier Transform (FT). Because the CTFT deals with nonperiodic signals, we must find a way to include all real frequencies in the general equations. For the CTFT we simply utilize integration over real numbers rather than summation over integers in order to express the aperiodic signals.

Fourier Transform Synthesis

Joseph Fourier demonstrated that an arbitrary $s(t)$ can be written as a linear combination of harmonic complex sinusoids
Equation:

$$s(t) = \sum_{n=-\infty}^{\infty} c_n e^{j\omega_0 nt}$$

where $\omega_0 = \frac{2\pi}{T}$ is the fundamental frequency. For almost all $s(t)$ of practical interest, there exists c_n to make [link] true. If $s(t)$ is finite energy ($s(t) \in L^2[0,T]$), then the equality in [link] holds in the sense of energy convergence; if $s(t)$ is continuous, then [link] holds pointwise. Also, if $s(t)$ meets some mild conditions (the Dirichlet conditions), then [link] holds pointwise everywhere except at points of discontinuity.

The c_n - called the Fourier coefficients - tell us "how much" of the sinusoid $e^{j\omega_0 nt}$ is in $s(t)$. The formula shows $s(t)$ as a sum of complex exponentials, each of which is easily processed by an LTI system (since it is an eigenfunction of **every** LTI system). Mathematically, it tells us that the set of complex exponentials $\left\{\forall n, n \in \mathbb{Z} : \left(e^{j\omega_0 nt}\right)\right\}$ form a basis for the space of T-periodic continuous time functions.

Equations

Now, in order to take this useful tool and apply it to arbitrary non-periodic signals, we will have to delve deeper into the use of the superposition principle. Let $s_T(t)$ be a periodic signal having period T. We want to

consider what happens to this signal's spectrum as the period goes to infinity. We denote the spectrum for any assumed value of the period by $c_n\left(T\right)$. We calculate the spectrum according to the Fourier formula for a periodic signal, known as the Fourier Series (for more on this derivation, see the section on **Fourier Series**.)
Equation:

$$c_n = \frac{1}{T}\int_0^T s\left(t\right)\,\exp\left(-\imath\omega_0 t\right)\,dt$$

where $\omega_0 = \frac{2\pi}{T}$ and where we have used a symmetric placement of the integration interval about the origin for subsequent derivational convenience. We vary the frequency index n proportionally as we increase the period. Define
Equation:

$$S_T\left(f\right) \equiv Tc_n = \frac{1}{T}\int_0^T \left(S_T\left(f\right)\,\exp\left(\imath\omega_0 t\right)\,dt$$

making the corresponding Fourier Series
Equation:

$$s_T\left(t\right) = \sum_{-\infty}^{\infty} f\left(t\right)\,\exp\left(\imath\omega_0 t\right)\,\frac{1}{T}\Bigg)$$

As the period increases, the spectral lines become closer together, becoming a continuum. Therefore,
Equation:

$$\lim_{T\to\infty}\,s_T\left(t\right) \equiv s\left(t\right) = \int_{-\infty}^{\infty} S\left(f\right)\,\exp\left(\imath\omega_0 t\right)\,df$$

with
Equation:

$$S\left(f\right)=\int_{-\infty}^{\infty}s\left(t\right)\,\exp\,\left(-\imath\omega_{0}t\right)dt$$

Equation:

Continuous-Time Fourier Transform

$$\mathscr{F}(\Omega)=\int_{-\infty}^{\infty}f(t)e^{-(i\Omega t)}\,\mathrm{d}\,t$$

Equation:

Inverse CTFT

$$f(t)=\frac{1}{2\pi}\int_{-\infty}^{\infty}\mathscr{F}(\Omega)e^{i\Omega t}\,\mathrm{d}\,\Omega$$

Note: It is not uncommon to see the above formula written slightly different. One of the most common differences is the way that the exponential is written. The above equations use the radial frequency variable Ω in the exponential, where $\Omega=2\pi f$, but it is also common to include the more explicit expression, $i2\pi ft$, in the exponential. Click here for an overview of the notation used in Connexion's DSP modules.

Example:
We know from Euler's formula that
$\cos\left(\omega t\right)+\sin\left(\omega t\right)=\frac{1-j}{2}e^{j\omega t}+\frac{1+j}{2}e^{-j\omega t}$.

CTFT Definition Demonstration



Interact (when online) with a Mathematica CDF demonstrating Continuous Time Fourier Transform. To Download, right-click and save as .cdf.

Example Problems

Exercise:

Problem: Find the Fourier Transform (CTFT) of the function
Equation:

$$f(t) = \begin{cases} e^{-(\alpha t)} & \text{if } t \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Solution:

In order to calculate the Fourier transform, all we need to use is [link], complex exponentials, and basic calculus.
Equation:

$$\begin{aligned}\mathscr{F}(\Omega) &= \int_{-\infty}^{\infty} f(t) e^{-(i\Omega t)} \, \mathrm{d}\, t \\ &= \int_{0}^{\infty} e^{-(\alpha t)} e^{-(i\Omega t)} \, \mathrm{d}\, t \\ &= \int_{0}^{\infty} e^{(-t)(\alpha + i\Omega)} \, \mathrm{d}\, t \\ &= 0 - \tfrac{-1}{\alpha + i\Omega}\end{aligned}$$

Equation:

$$\mathscr{F}(\Omega) = \frac{1}{\alpha + i\Omega}$$

Exercise:

Problem:

Find the inverse Fourier transform of the ideal lowpass filter defined by

Equation:

$$X(\Omega) = \begin{cases} 1 & \text{if } \ |\Omega| \leq \mathrm{M} \\ 0 & \text{otherwise} \end{cases}$$

Solution:

Here we will use [link] to find the inverse FT given that $t \neq 0$.

Equation:

$$\begin{aligned} x(t) &= \tfrac{1}{2\pi} \int_{-M}^{M} e^{i(\Omega, t)} \, \mathrm{d}\, \Omega \\ &= \tfrac{1}{2\pi} e^{i(\Omega, t)} \Big|_{\Omega, \Omega = e^{iw}} \\ &= \tfrac{1}{\pi t} \sin(Mt) \end{aligned}$$

Equation:

$$x(t) = \frac{M}{\pi} \left(\operatorname{sinc} \frac{Mt}{\pi} \right)$$

Fourier Transform Summary

Because complex exponentials are eigenfunctions of LTI systems, it is often useful to represent signals using a set of complex exponentials as a basis. The continuous time Fourier series synthesis formula expresses a continuous time, periodic function as the sum of continuous time, discrete frequency complex exponentials.
Equation:

$$f(t) = \sum_{n=-\infty}^{\infty} c_n e^{j\omega_0 n t}$$

The continuous time Fourier series analysis formula gives the coefficients of the Fourier series expansion.
Equation:

$$c_n = \frac{1}{T} \int_0^T f(t) e^{-(j\omega_0 n t)} \, \mathrm{d}\, t$$

In both of these equations $\omega_0 = \frac{2\pi}{T}$ is the fundamental frequency.

Discrete-Time Fourier Transform (DTFT)
Discussion of Discrete-time Fourier Transforms. Topics include comparison with analog transforms and discussion of Parseval's theorem.

The Fourier transform of the discrete-time signal $s(n)$ is defined to be
Equation:

$$S\left(e^{i2\pi f}\right) = \sum_{n=-\infty}^{\infty} s(n) e^{-(i2\pi f n)}$$

Frequency here has no units. As should be expected, this definition is linear, with the transform of a sum of signals equaling the sum of their transforms. Real-valued signals have conjugate-symmetric spectra:
$S\left(e^{-(i2\pi f)}\right) = S(e^{j2\pi f})$.
Exercise:

Problem:

A special property of the discrete-time Fourier transform is that it is periodic with period one: $S\left(e^{i2\pi(f+1)}\right) = S\left(e^{i2\pi f}\right)$. Derive this property from the definition of the DTFT.

Solution:
Equation:

$$\begin{aligned} S\left(e^{i2\pi(f+1)}\right) &= \sum_{n=-\infty}^{\infty} s(n) e^{-(i2\pi(f+1)n)} \\ &= \sum_{n=-\infty}^{\infty} e^{-(i2\pi n)} s(n) e^{-(i2\pi f n)} \\ &= \sum_{n=-\infty}^{\infty} s(n) e^{-(i2\pi f n)} \\ &= S\left(e^{i2\pi f}\right) \end{aligned}$$

Because of this periodicity, we need only plot the spectrum over one period to understand completely the spectrum's structure; typically, we plot the spectrum over the frequency range $\left[-\frac{1}{2}, \frac{1}{2}\right]$. When the signal is real-

valued, we can further simplify our plotting chores by showing the spectrum only over $\left[0, \frac{1}{2}\right]$; the spectrum at negative frequencies can be derived from positive-frequency spectral values.

When we obtain the discrete-time signal via sampling an analog signal, the Nyquist frequency corresponds to the discrete-time frequency $\frac{1}{2}$. To show this, note that a sinusoid having a frequency equal to the Nyquist frequency $\frac{1}{2T_s}$ has a sampled waveform that equals

$$\cos\left(2\pi \times \frac{1}{2Ts} nTs\right) = \cos(\pi n) = (-1)^n$$

The exponential in the DTFT at frequency $\frac{1}{2}$ equals $e^{-\frac{i2\pi n}{2}} = e^{-(i\pi n)} = (-1)^n$, meaning that discrete-time frequency equals analog frequency multiplied by the sampling interval
Equation:

$$f_D = f_A T_s$$

f_D and f_A represent discrete-time and analog frequency variables, respectively. The aliasing figure provides another way of deriving this result. As the duration of each pulse in the periodic sampling signal $p_{T_s}(t)$ narrows, the amplitudes of the signal's spectral repetitions, which are governed by the Fourier series coefficients of $p_{T_s}(t)$, become increasingly equal. Examination of the periodic pulse signal reveals that as Δ decreases, the value of c_0, the largest Fourier coefficient, decreases to zero: $|c_0| = \frac{A\Delta}{T_s}$. Thus, to maintain a mathematically viable Sampling Theorem, the amplitude A must increase as $\frac{1}{\Delta}$, becoming infinitely large as the pulse duration decreases. Practical systems use a small value of Δ, say $0.1 \cdot T_s$ and use amplifiers to rescale the signal. Thus, the sampled signal's spectrum becomes periodic with period $\frac{1}{T_s}$. Thus, the Nyquist frequency $\frac{1}{2T_s}$ corresponds to the frequency $\frac{1}{2}$.

Example:
Let's compute the discrete-time Fourier transform of the exponentially decaying sequence $s(n) = a^n u(n)$, where $u(n)$ is the unit-step sequence. Simply plugging the signal's expression into the Fourier transform formula,
Equation:

$$\begin{aligned} S\left(e^{i2\pi f}\right) &= \sum_{n=-\infty}^{\infty} a^n u(n) e^{-(i2\pi f n)} \\ &= \sum_{n=0}^{\infty} \left(a e^{-(i2\pi f)}\right)^n \end{aligned}$$

This sum is a special case of the **geometric series**.
Equation:

$$\sum_{n=0}^{\infty} \alpha^n = \forall \alpha, |\alpha| < 1 : \left(\frac{1}{1-\alpha}\right)$$

Thus, as long as $|a| < 1$, we have our Fourier transform.
Equation:

$$S\left(e^{i2\pi f}\right) = \frac{1}{1 - a e^{-(i2\pi f)}}$$

Using Euler's relation, we can express the magnitude and phase of this spectrum.
Equation:

$$S\left(e^{i2\pi f}\right) = \frac{1}{\sqrt{\left(1 - a\cos(2\pi f)\right)^2 + a^2 \sin^2(2\pi f)}}$$

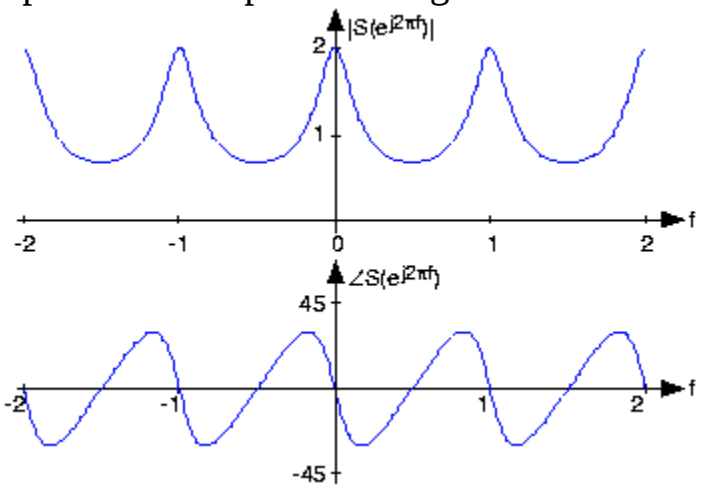
Equation:

$$\angle\left(S\left(e^{i2\pi f}\right)\right) = -\tan^{-1}\left(\frac{a\sin(2\pi f)}{1 - a\cos(2\pi f)}\right)$$

No matter what value of a we choose, the above formulae clearly demonstrate the periodic nature of the spectra of discrete-time signals. [link] shows indeed that the spectrum is a periodic function. We need only

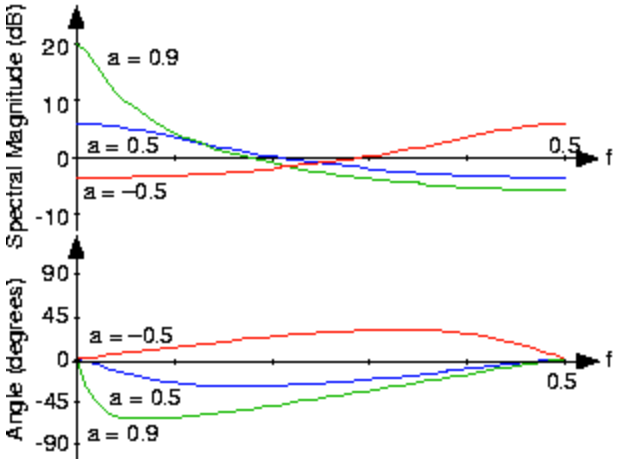
consider the spectrum between $-\frac{1}{2}$ and $\frac{1}{2}$ to unambiguously define it. When $a > 0$, we have a lowpass spectrum—the spectrum diminishes as frequency increases from 0 to $\frac{1}{2}$—with increasing a leading to a greater low frequency content; for $a < 0$, we have a highpass spectrum ([link]).

Spectrum of exponential signal



The spectrum of the exponential signal ($a = 0.5$) is shown over the frequency range [-2, 2], clearly demonstrating the periodicity of all discrete-time spectra. The angle has units of degrees.

Spectra of exponential signals



The spectra of several exponential signals are shown. What is the apparent relationship between the spectra for $a = 0.5$ and $a = -0.5$?

Example:
Analogous to the analog pulse signal, let's find the spectrum of the length-N pulse sequence.
Equation:

$$s(n) = \begin{cases} 1 \text{ if } 0 \leq n \leq N-1 \\ 0 \text{ otherwise} \end{cases}$$

The Fourier transform of this sequence has the form of a truncated geometric series.
Equation:

$$S\left(e^{i2\pi f}\right) = \sum_{n=0}^{N-1} e^{-(i2\pi fn)}$$

For the so-called finite geometric series, we know that
Equation:

$$\sum_{n=n_0}^{N+n_0-1} \alpha^n = \alpha^{n_0} \frac{1-\alpha^N}{1-\alpha}$$

for **all** values of α.

Exercise:

Problem:

Derive this formula for the finite geometric series sum. The "trick" is to consider the difference between the series' sum and the sum of the series multiplied by α.

Solution:

$$\alpha \sum_{n=n_0}^{N+n_0-1} \alpha^n - \sum_{n=n_0}^{N+n_0-1} \alpha^n = \alpha^{N+n_0} - \alpha^{n_0}$$

which, after manipulation, yields the geometric sum formula.

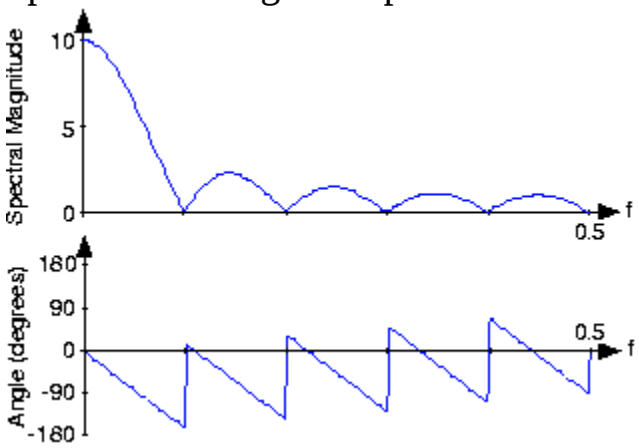
Applying this result yields ([link].)
Equation:

$$\begin{aligned} S\left(e^{i2\pi f}\right) &= \frac{1-e^{-(i2\pi fN)}}{1-e^{-(i2\pi f)}} \\ &= e^{-(i\pi f(N-1))} \frac{\sin(\pi fN)}{\sin(\pi f)} \end{aligned}$$

The ratio of sine functions has the generic form of $\frac{\sin(Nx)}{\sin(x)}$, which is known as the **discrete-time sinc function** $\operatorname{dsinc}(x)$. Thus, our transform can be concisely expressed as $S\left(e^{i2\pi f}\right) = e^{-(i\pi f(N-1))} \operatorname{dsinc}(\pi f)$. The discrete-

time pulse's spectrum contains many ripples, the number of which increase with N, the pulse's duration.

Spectrum of length-ten pulse



The spectrum of a length-ten pulse is shown. Can you explain the rather complicated appearance of the phase?

The inverse discrete-time Fourier transform is easily derived from the following relationship:

Equation:

$$\begin{aligned} \int_{-\frac{1}{2}}^{\frac{1}{2}} e^{-(i2\pi fm)} e^{i2\pi fn} \, \mathrm{d}\, f &= \begin{cases} 1 & \text{if } m = n \\ 0 & \text{if } m \neq n \end{cases} \\ &= \delta(m-n) \end{aligned}$$

Therefore, we find that

Equation:

$$\begin{aligned}\int_{-\frac{1}{2}}^{\frac{1}{2}} S\left(e^{i2\pi f}\right)e^{i2\pi fn}\ \mathrm{d}\ f &= \int_{-\frac{1}{2}}^{\frac{1}{2}} \sum_{mm} s(m)e^{-(i2\pi fm)}e^{i2\pi fn}\ \mathrm{d}\ f \\ &= \sum_{mm} s(m)\int_{-\frac{1}{2}}^{\frac{1}{2}} e^{(-(i2\pi f))(m-n)}\ \mathrm{d}\ f \\ &= s(n)\end{aligned}$$

The Fourier transform pairs in discrete-time are
Equation:

$$S\left(e^{i2\pi f}\right) = \sum_{n=-\infty}^{\infty} s(n)e^{-(i2\pi fn)}$$
$$s(n) = \int_{-\frac{1}{2}}^{\frac{1}{2}} S\left(e^{i2\pi f}\right)e^{i2\pi fn}\ \mathrm{d}\ f$$

The properties of the discrete-time Fourier transform mirror those of the analog Fourier transform. The DTFT properties table shows similarities and differences. One important common property is Parseval's Theorem.
Equation:

$$\sum_{n=-\infty}^{\infty} \left(|s(n)|\right)^2 = \int_{-\frac{1}{2}}^{\frac{1}{2}} \left(\ S\left(e^{i2\pi f}\right)\ \right)^2\ \mathrm{d}\ f$$

To show this important property, we simply substitute the Fourier transform expression into the frequency-domain expression for power.
Equation:

$$\begin{aligned}\int_{-\frac{1}{2}}^{\frac{1}{2}} \left(\ S\left(e^{i2\pi f}\right)\ \right)^2\ \mathrm{d}\ f &= \int_{-\frac{1}{2}}^{\frac{1}{2}} \sum_{nn} s(n)e^{-(i2\pi fn)} \sum_{mm} s(n)e^{i2\pi fm}\ \mathrm{d}\ f \\ &= \sum_{n,mn,m} s(n)s(n)\int_{-\frac{1}{2}}^{\frac{1}{2}} e^{i2\pi f(m-n)}\ \mathrm{d}\ f\end{aligned}$$

Using the orthogonality relation, the integral equals $\delta(m-n)$, where $\delta(n)$ is the unit sample. Thus, the double sum collapses into a single sum because nonzero values occur only when $n = m$, giving Parseval's Theorem as a result. We term $\sum_{nn} s^2(n)$ the energy in the discrete-time signal $s(n)$ in

spite of the fact that discrete-time signals don't consume (or produce for that matter) energy. This terminology is a carry-over from the analog world.

Exercise:

Problem:

Suppose we obtained our discrete-time signal from values of the product $s(t)p_{T_s}(t)$, where the duration of the component pulses in $p_{T_s}(t)$ is Δ. How is the discrete-time signal energy related to the total energy contained in $s(t)$? Assume the signal is bandlimited and that the sampling rate was chosen appropriate to the Sampling Theorem's conditions.

Solution:

If the sampling frequency exceeds the Nyquist frequency, the spectrum of the samples equals the analog spectrum, but over the normalized analog frequency fT. Thus, the energy in the sampled signal equals the original signal's energy multiplied by T.

DFT as a Matrix Operation

Matrix Review

Recall:

- Vectors in $\mathbb{R}^N$:

$$\forall x_i, x_i \in \mathbb{R}: \quad \boldsymbol{x} = \begin{matrix} x_0 \\ x_1 \\ \dots \\ x_{N-1} \end{matrix}$$

- Vectors in $\mathbb{C}^N$:

$$\forall x_i, x_i \in \mathbb{C}: \quad \boldsymbol{x} = \begin{matrix} x_0 \\ x_1 \\ \dots \\ x_{N-1} \end{matrix}$$

- Transposition:

    1. transpose:

$$\boldsymbol{x}^T = \begin{pmatrix} x_0 & x_1 & \dots & x_{N-1} \end{pmatrix}$$

    2. conjugate:

$$\boldsymbol{x}^{\mathrm{H}} = \begin{pmatrix} x_0 & x_1 & \dots & x_{N-1} \end{pmatrix}$$

- Inner product:

    1. real:

$$\boldsymbol{x}^T \boldsymbol{y} = \sum_{i=0}^{N-1} x_i y_i$$

2. complex:

$$\boldsymbol{x}^{\mathrm{H}}\boldsymbol{y} = \sum_{i=0}^{N-1} x_n y_n$$

- Matrix Multiplication:

$$A\boldsymbol{x} = \begin{matrix} a_{00} & a_{01} & \dots & a_{0,N-1} \\ a_{10} & a_{11} & \dots & a_{1,N-1} \\ \vdots & \vdots & \dots & \vdots \\ a_{N-1,0} & a_{N-1,1} & \dots & a_{N-1,N-1} \end{matrix} \begin{matrix} x_0 \\ x_1 \\ \dots \\ x_{N-1} \end{matrix} = \begin{matrix} y_0 \\ y_1 \\ \dots \\ y_{N-1} \end{matrix}$$

$$y_k = \sum_{n=0}^{N-1} a_{kn} x_n$$

- Matrix Transposition:

$$A^T = \begin{matrix} a_{00} & a_{10} & \dots & a_{N-1,0} \\ a_{01} & a_{11} & \dots & a_{N-1,1} \\ \vdots & \vdots & \dots & \vdots \\ a_{0,N-1} & a_{1,N-1} & \dots & a_{N-1,N-1} \end{matrix}$$

Matrix transposition involved simply swapping the rows with columns.

$$A^{\mathrm{H}} = A^T$$

The above equation is Hermitian transpose.

$$\boldsymbol{A}^T{}_{k,n} = \boldsymbol{A}_{n,k}$$

$$A^{\mathrm{H}}{}_{k,n} = \boldsymbol{A}_{\,n,k}$$

Representing DFT as Matrix Operation

Now let's represent the DFT in vector-matrix notation.

$$\boldsymbol{x} = \begin{matrix} x[0] \\ x[1] \\ \cdots \\ x[N-1] \end{matrix}$$

$$\boldsymbol{X} = \begin{matrix} X[0] \\ X[1] \\ \cdots \\ X[N-1] \end{matrix} \quad \in \mathbb{C}^N$$

Here $\boldsymbol{x}$ is the vector of time samples and $\boldsymbol{X}$ is the vector of DFT coefficients. How are $\boldsymbol{x}$ and $\boldsymbol{X}$ related:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-(i\frac{2\pi}{N}kn)}$$

where

$$a_{kn} = \left(e^{-(i\frac{2\pi}{N})}\right)^{kn} = {W_N}^{kn}$$

so

$$\boldsymbol{X} = W\boldsymbol{x}$$

where $\boldsymbol{X}$ is the DFT vector, W is the matrix and $\boldsymbol{x}$ the time domain vector.

$$W_{k,n} = \left(e^{-(i\frac{2\pi}{N})}\right)^{kn}$$

$$\boldsymbol{X} = W \begin{matrix} x[0] \\ x[1] \\ \cdots \\ x[N-1] \end{matrix}$$

IDFT:

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] \left(e^{i \frac{2\pi}{N}} \right)^{nk}$$

where

$$\left(e^{i \frac{2\pi}{N}} \right)^{nk} = {W_N}^{nk}$$

${W_N}^{nk}$ is the matrix Hermitian transpose. So,

$$\boldsymbol{x} = \frac{1}{N} W^{\mathrm{H}} \boldsymbol{X}$$

where $\boldsymbol{x}$ is the time vector, $\frac{1}{N} W^{\mathrm{H}}$ is the inverse DFT matrix, and $\boldsymbol{X}$ is the DFT vector.

Introduction

Contents of Sampling chapter

- Introduction(Current module)
- Proof
- Illustrations
- Matlab Example
- Hold operation
- System view
- Aliasing applet
- Exercises
- Table of formulas

Why sample?

This section introduces sampling. Sampling is the necessary fundament for all digital signal processing and communication. Sampling can be defined as the process of measuring an analog signal at distinct points.

Digital representation of analog signals offers advantages in terms of

- robustness towards noise, meaning we can send more bits/s
- use of flexible processing equipment, in particular the computer
- more reliable processing equipment
- easier to adapt complex algorithms

Claude E. Shannon



Claude
Elwood
Shannon
(1916-2001)

Claude Shannon has been called the father of information theory, mainly due to his landmark papers on the "Mathematical theory of communication". Harry Nyquist was the first to state the sampling theorem in 1928, but it was not proven until Shannon proved it 21 years later in the paper "Communications in the presence of noise".

Notation

In this chapter we will be using the following notation

- Original analog signal $x(t)$
- Sampling frequency F_s
- Sampling interval T_s (Note that: $F_s = \frac{1}{T_s}$)
- Sampled signal $x_s(n)$. (Note that $x_s(n) = x(nT_s)$)
- Real angular frequency Ω
- Digital angular frequency ω. (Note that: $\omega = \Omega T_s$)

The Sampling Theorem

Note: When sampling an analog signal the sampling frequency must be greater than twice the highest frequency component of the analog signal to be able to reconstruct the original signal from the sampled version.

Finished? Have at look at: Proof Illustrations Matlab Example Aliasing applet Hold operation System view Exercises

Proof

Note: In order to recover the signal $x(t)$ from it's samples exactly, it is necessary to sample $x(t)$ at a rate greater than twice it's highest frequency component.

Introduction

As mentioned earlier, sampling is the necessary fundament when we want to apply digital signal processing on analog signals.

Here we present the proof of the sampling theorem. The proof is divided in two. First we find an expression for the spectrum of the signal resulting from sampling the original signal $x(t)$. Next we show that the signal $x(t)$ can be recovered from the samples. Often it is easier using the frequency domain when carrying out a proof, and this is also the case here.

Key points in the proof

- We find an equation for the spectrum of the sampled signal
- We find a simple method to reconstruct the original signal
- The sampled signal has a periodic spectrum...
- ...and the period is $2 \times \pi F_s$

Proof part 1 - Spectral considerations

By sampling $x(t)$ every T_s second we obtain $x_s(n)$. The inverse fourier transform of this time discrete signal is

Equation:

$$x_s(n) = \frac{1}{2\pi} \int_{-\pi}^{\pi} X_s\left(e^{i\omega}\right) e^{i\omega n} \, \mathrm{d}\, \omega$$

For convenience we express the equation in terms of the real angular frequency Ω using $\omega = \Omega T_s$. We then obtain
Equation:

$$x_s(n) = \frac{T_s}{2\pi} \int_{\frac{-\pi}{T_s}}^{\frac{\pi}{T_s}} X_s\left(e^{i\Omega T_s}\right) e^{i\Omega T_s n} \, \mathrm{d}\, \Omega$$

The inverse fourier transform of a continuous signal is
Equation:

$$x(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} X(i\Omega) e^{i\Omega t} \, \mathrm{d}\, \Omega$$

From this equation we find an expression for $x\left(nT_s\right)$
Equation:

$$x(nT_s) = \frac{1}{2\pi} \int_{-\infty}^{\infty} X(i\Omega) e^{i\Omega n T_s} \, \mathrm{d}\, \Omega$$

To account for the difference in region of integration we split the integration in [link] into subintervals of length $\frac{2\pi}{T_s}$ and then take the sum over the resulting integrals to obtain the complete area.
Equation:

$$x(nT_s) = \frac{1}{2\pi} \sum_{k=-\infty}^{\infty} \int_{\frac{(2k-1)\pi}{T_s}}^{\frac{(2k+1)\pi}{T_s}} X(i\Omega) e^{i\Omega n T_s} \, \mathrm{d}\, \Omega$$

Then we change the integration variable, setting $\Omega = \eta + \frac{2 \times \pi k}{T_s}$
Equation:

$$x(nT_s) = \frac{1}{2\pi} \sum_{k=-\infty}^{\infty} \int_{\frac{-\pi}{T_s}}^{\frac{\pi}{T_s}} X\left(i\left(\eta + \frac{2 \times \pi k}{T_s}\right)\right) e^{i\left(\eta + \frac{2 \times \pi k}{T_s}\right) n T_s} \, \mathrm{d}\, \eta$$

We obtain the final form by observing that $e^{i2 \times \pi k n} = 1$, reinserting $\eta = \Omega$ and multiplying by $\frac{T_s}{T_s}$

Equation:

$$x(nT_s) = \frac{T_s}{2\pi} \int_{\frac{-\pi}{T_s}}^{\frac{\pi}{T_s}} \sum_{k=-\infty}^{\infty} \frac{1}{T_s} X\left(i\left(\Omega + \frac{2 \times \pi k}{T_s}\right)\right) e^{i \Omega n T_s} \, \mathrm{d}\, \Omega$$

To make $x_s(n) = x(nT_s)$ for all values of n, the integrands in [link] and [link] have to agreee, that is

Equation:

$$X_s\left(e^{i \Omega T_s}\right) = \frac{1}{T_s} \sum_{k=-\infty}^{\infty} X\left(i\left(\Omega + \frac{2\pi k}{T_s}\right)\right)$$

This is a central result. We see that the digital spectrum consists of a sum of shifted versions of the original, analog spectrum. Observe the periodicity!

We can also express this relation in terms of the digital angular frequency $\omega = \Omega T_s$

Equation:

$$X_s\left(e^{i \omega}\right) = \frac{1}{T_s} \sum_{k=-\infty}^{\infty} X\left(i\frac{\omega + 2 \times \pi k}{T_s}\right)$$

This concludes the first part of the proof. Now we want to find a reconstruction formula, so that we can recover $x(t)$ from $x_s(n)$.

Proof part II - Signal reconstruction

For a bandlimited signal the inverse fourier transform is
Equation:

$$x(t) = \frac{1}{2\pi} \int_{\frac{-\pi}{T_s}}^{\frac{\pi}{T_s}} X(i\Omega) e^{i\Omega t} \, \mathrm{d}\, \Omega$$

In the interval we are integrating we have: $X_s\left(e^{i\Omega T_s}\right) = \frac{X(i\Omega)}{T_s}$.
Substituting this relation into [link] we get
Equation:

$$x(t) = \frac{T_s}{2\pi} \int_{\frac{-\pi}{T_s}}^{\frac{\pi}{T_s}} X_s\left(e^{i\Omega T_s}\right) e^{i\Omega t} \, \mathrm{d}\, \Omega$$

Using the DTFT relation for $X_s\left(e^{i\Omega T_s}\right)$ we have
Equation:

$$x(t) = \frac{T_s}{2\pi} \int_{\frac{-\pi}{T_s}}^{\frac{\pi}{T_s}} \sum_{n=-\infty}^{\infty} x_s(n) e^{-(i\Omega n T_s)} e^{i\Omega t} \, \mathrm{d}\, \Omega$$

Interchanging integration and summation (under the assumption of convergence) leads to
Equation:

$$x(t) = \frac{T_s}{2\pi} \sum_{n=-\infty}^{\infty} x_s(n) \int_{\frac{-\pi}{T_s}}^{\frac{\pi}{T_s}} e^{i\Omega(t - nT_s)} \, \mathrm{d}\, \Omega$$

Finally we perform the integration and arrive at the important reconstruction formula
Equation:

$$x(t) = \sum_{n=-\infty}^{\infty} x_s(n) \frac{\sin\left(\frac{\pi}{T_s}\left(t - nT_s\right)\right)}{\frac{\pi}{T_s}\left(t - nT_s\right)}$$

(Thanks to R.Loos for pointing out an error in the proof.)

Summary

Note: $X_s\left(e^{i\Omega T_s}\right) = \frac{1}{T_s}\sum_{k=-\infty}^{\infty} X\left(i\left(\Omega + \frac{2\pi k}{T_s}\right)\right)$

Note: $x(t) = \sum_{n=-\infty}^{\infty} x_s(n) \frac{\sin\left(\frac{\pi}{T_s}(t-nT_s)\right)}{\frac{\pi}{T_s}(t-nT_s)}$

Go to Introduction Illustrations Matlab Example Hold operation Aliasing applet System view Exercises ?

Illustrations

In this module we illustrate the processes involved in sampling and reconstruction. To see how all these processes work together as a whole, take a look at the system view. In Sampling and reconstruction with Matlab we provide a Matlab script for download. The matlab script shows the process of sampling and reconstruction **live**.

Basic examples

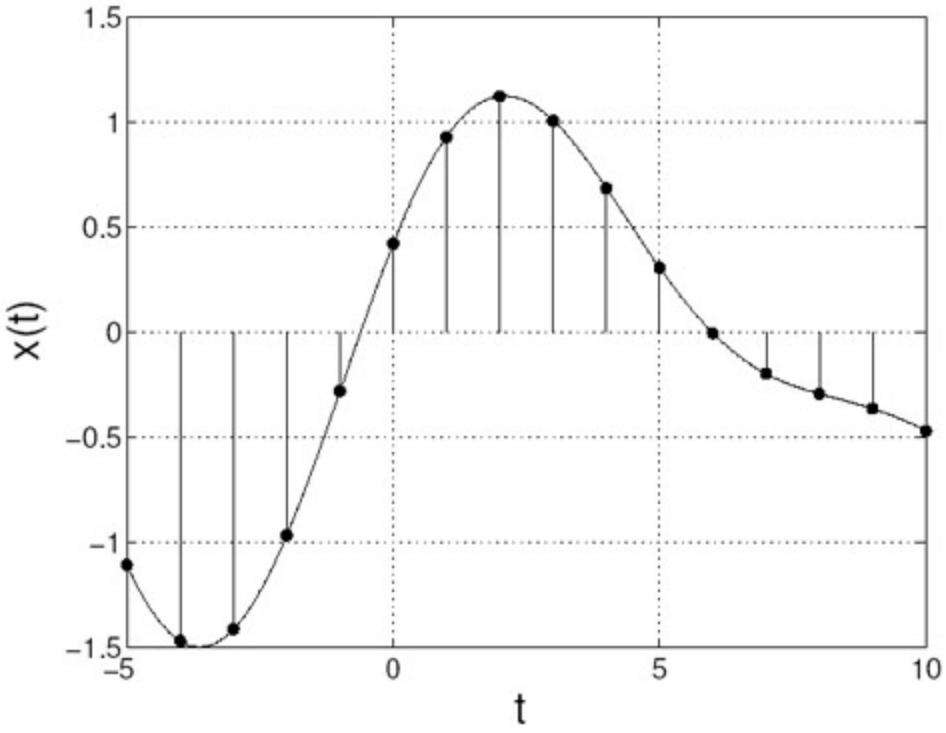
Example:
To sample an analog signal with 3000 Hz as the highest frequency component requires sampling at 6000 Hz or above.

Example:
The sampling theorem can also be applied in two dimensions, i.e. for image analysis. A 2D sampling theorem has a simple physical interpretation in image analysis: Choose the sampling interval such that it is less than or equal to half of the smallest interesting detail in the image.

The process of sampling

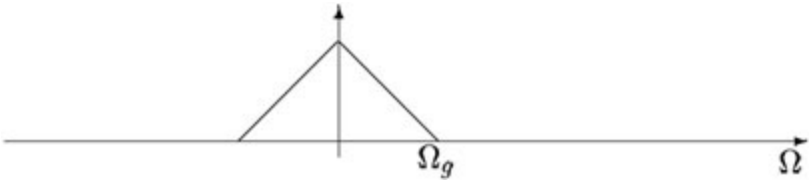
We start off with an analog signal. This can for example be the sound coming from your stereo at home or your friend talking.

The signal is then sampled uniformly. Uniform sampling implies that we sample every T_s seconds. In [link] we see an analog signal. The analog signal has been sampled at times $t = nT_s$.



Analog signal, samples are marked with dots.

In signal processing it is often more convenient and easier to work in the frequency domain. So let's look at at the signal in frequency domain, [link]. For illustration purposes we take the frequency content of the signal as a triangle. (If you Fourier transform the signal in [link] you will not get such a nice triangle.)



The spectrum $X(i\Omega)$.

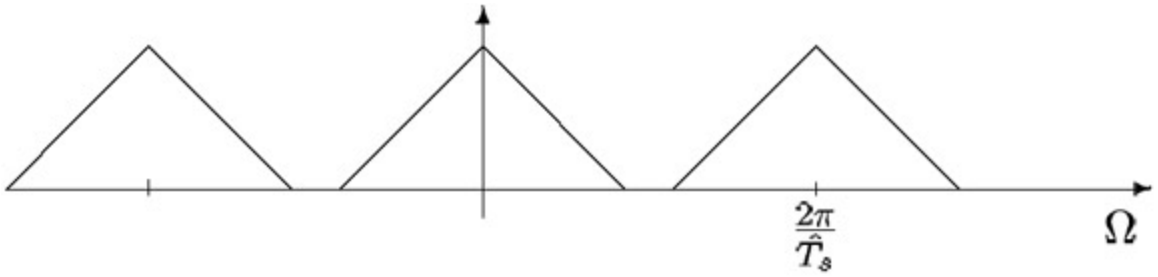
Notice that the signal in [link] is bandlimited. We can see that the signal is bandlimited because $X(i\Omega)$ is zero outside the interval $[-\Omega_g, \Omega_g]$. Equivalentely we can state that the signal has no angular frequencies above Ω_g, corresponding to no frequencies above $F_g = \frac{\Omega_g}{2\pi}$.

Now let's take a look at the sampled signal in the frequency domain. While proving the sampling theorem we found the the spectrum of the sampled signal consists of a sum of shifted versions of the analog spectrum. Mathematically this is described by the following equation:
Equation:

$$X_s\left(e^{i\Omega T_s}\right) = \frac{1}{T_s} \sum_{k=-\infty}^{\infty} X\left(i\left(\Omega + \frac{2\pi k}{T_s}\right)\right)$$

Sampling fast enough

In [link] we show the result of sampling $x(t)$ according to the sampling theorem. This means that when sampling the signal in [link]/[link] we use $F_s \geq 2F_g$. Observe in [link] that we have the same spectrum as in [link] for $\Omega \in$ [-Ω_g, Ω_g], except for the scaling factor $\frac{1}{T_s}$. This is a consequence of the sampling frequency. As mentioned in the proof the spectrum of the sampled signal is periodic with period $2\pi F_s = \frac{2\pi}{T_s}$.



The spectrum X_s. Sampling frequency is OK.

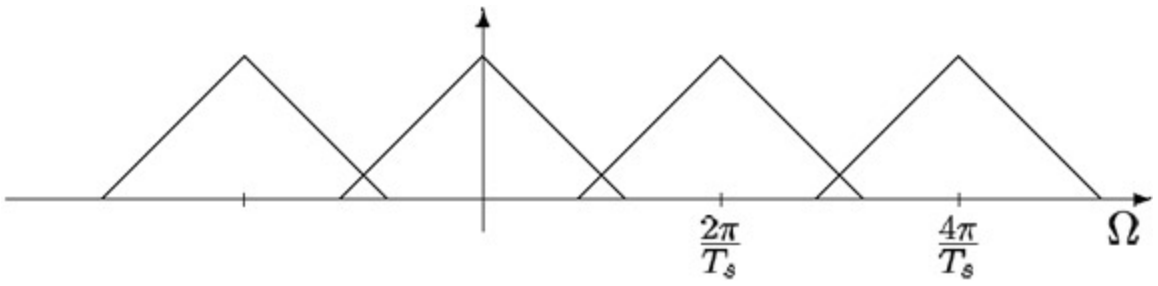
So now we are, according to the sample theorem, able to reconstruct the original signal **exactly**. How we can do this will be explored further down under reconstruction. But first we will take a look at what happens when we sample too slowly.

Sampling too slowly

If we sample $x(t)$ too slowly, that is $F_s < 2F_g$, we will get overlap between the repeated spectra, see [link]. According to [link] the resulting spectra is the sum of these. This overlap gives rise to the concept of aliasing.

> **Note:** If the sampling frequency is less than twice the highest frequency component, then frequencies in the original signal that are above half the sampling rate will be "aliased" and will appear in the resulting signal as lower frequencies.

The consequence of aliasing is that we cannot recover the original signal, so aliasing has to be avoided. Sampling too slowly will produce a sequence $x_s(n)$ that could have orginated from a number of signals. So there is **no** chance of recovering the original signal. To learn more about aliasing, take a look at this module. (Includes an applet for demonstration!)



The spectrum X_s. Sampling frequency is too low.

To avoid aliasing we have to sample fast enough. But if we can't sample fast enough (possibly due to costs) we can include an Anti-Aliasing filter. This will not able us to get an exact reconstruction but can still be a good solution.

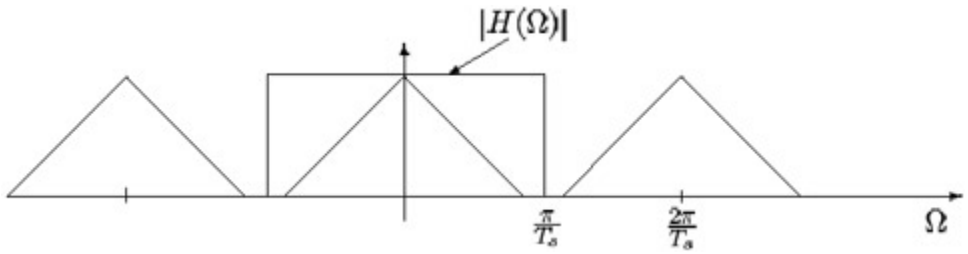
Note: Typically a low-pass filter that is applied before sampling to ensure that no components with frequencies greater than half the sample frequency remain.

Example:
The stagecoach effect
In older western movies you can observe aliasing on a stagecoach when it starts to roll. At first the spokes appear to turn forward, but as the stagecoach increase its speed the spokes appear to turn backward. This comes from the fact that the sampling rate, here the number of frames per second, is too low. We can view each frame as a sample of an image that is changing continuously in time. (Applet illustrating the stagecoach effect)

Reconstruction

Given the signal in [link] we want to recover the original signal, but the question is how?

When there is no overlapping in the spectrum, the spectral component given by $k = 0$ (see [link]),is equal to the spectrum of the analog signal. This offers an oppurtunity to use a simple reconstruction process. Remember what you have learned about filtering. What we want is to change signal in [link] into that of [link]. To achieve this we have to remove all the extra components generated in the sampling process. To remove the extra components we apply an ideal analog low-pass filter as shown in [link] As we see the ideal filter is rectangular in the frequency domain. A rectangle in the frequency domain corresponds to a sinc function in time domain (and vice versa).



$H(i\Omega)$ The ideal reconstruction filter.

Then we have reconstructed the original spectrum, and as we know **if two signals are identical in the frequency domain, they are also identical in the time domain**. End of reconstruction.

Conclusions

The Shannon sampling theorem requires that the input signal prior to sampling is band-limited to at most half the sampling frequency. Under this condition the samples give an exact signal representation. It is truly remarkable that such a broad and useful class signals can be represented that easily!

We also looked into the problem of reconstructing the signals form its samples. Again the simplicity of the **principle** is striking: linear filtering by an ideal low-pass filter will do the job. However, the ideal filter is impossible to create, but that is another story...

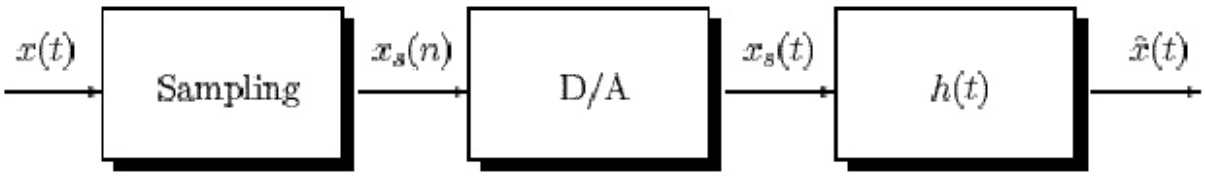
Go to? Introduction Proof Illustrations Matlab Example Aliasing applet Hold operation System view Exercises

Systems view of sampling and reconstruction

Ideal reconstruction system

[link] shows the ideal reconstruction system based on the results of the Sampling theorem proof.

[link] consists of a sampling device which produces a time-discrete sequence $x_s(n)$. The reconstruction filter, $h(t)$, is an ideal analog sinc filter, with $h(t) = \text{sinc}\left(\frac{t}{T_s}\right)$. We can't apply the time-discrete sequence $x_s(n)$ directly to the analog filter $h(t)$. To solve this problem we turn the sequence into an analog signal using delta functions. Thus we write $x_s(t) = \sum_{n=-\infty}^{\infty} x_s(n)\delta(t - nT)$.

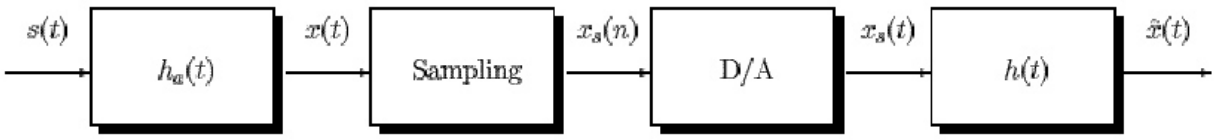


Ideal reconstruction system

But when will the system produce an output $\widehat{x}(t) = x(t)$? According to the sampling theorem we have $\widehat{x}(t) = x(t)$ when the sampling frequency, F_s, is at least twice the highest frequency component of $x(t)$.

Ideal system including anti-aliasing

To be sure that the reconstructed signal is free of aliasing it is customary to apply a lowpass filter, an anti-aliasing filter, before sampling as shown in [link].



Ideal reconstruction system with anti-aliasing filter

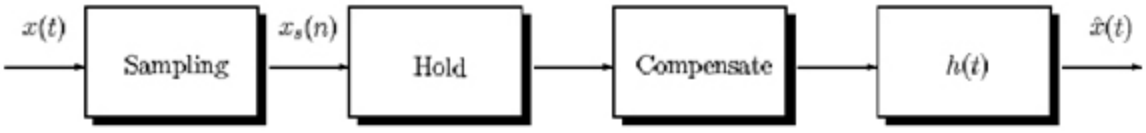
Again we ask the question of when the system will produce an output $\widehat{x}(t) = s(t)$? If the signal is entirely confined within the passband of the lowpass filter we will get perfect reconstruction if F_s is high enough.

But if the anti-aliasing filter removes the "higher" frequencies, (which in fact is the job of the anti-aliasing filter), we will **never** be able to **exactly** reconstruct the original signal, $s(t)$. If we sample fast enough we can reconstruct $x(t)$, which in most cases is satisfying.

The reconstructed signal, $\widehat{x}(t)$, will not have aliased frequencies. This is essential for further use of the signal.

Reconstruction with hold operation

To make our reconstruction system realizable there are many things to look into. Among them are the fact that any practical reconstruction system must input finite length pulses into the reconstruction filter. This can be accomplished by the hold operation. To alleviate the distortion caused by the hold opeator we apply the output from the hold device to a compensator. The compensation can be as accurate as we wish, this is cost and application consideration.



More practical reconstruction system with a hold component

By the use of the hold component the reconstruction will not be exact, but as mentioned above we can get as close as we want.

Introduction Proof Illustrations Matlab example Hold operation Aliasing applet Exercises

Sampling CT Signals: A Frequency Domain Perspective

Understanding Sampling in the Frequency Domain

We want to relate $x_c(t)$ directly to $x[n]$. Compute the CTFT of

$$x_s(t) = \sum_{n=-\infty}^{\infty} x_c(nT)\delta(t - nT)$$

Equation:

$$\begin{aligned} X_s(\Omega) &= \int_{-\infty}^{\infty} \sum_{n=-\infty}^{\infty} x_c(nT)\delta(t - nT)e^{(-i)\Omega t}\ \mathrm{d}\ t \\ &= \sum_{n=-\infty}^{\infty} x_c(nT) \int_{-\infty}^{\infty} \delta(t - nT)e^{(-i)\Omega t}\ \mathrm{d}\ t \\ &= \sum_{n=-\infty}^{\infty} x[n]e^{(-i)\Omega nT} \\ &= \sum_{n=-\infty}^{\infty} x[n]e^{(-i)\omega n} \\ &= X(\omega) \end{aligned}$$

where $\omega \equiv \Omega T$ and $X(\omega)$ is the DTFT of $x[n]$.

Note:

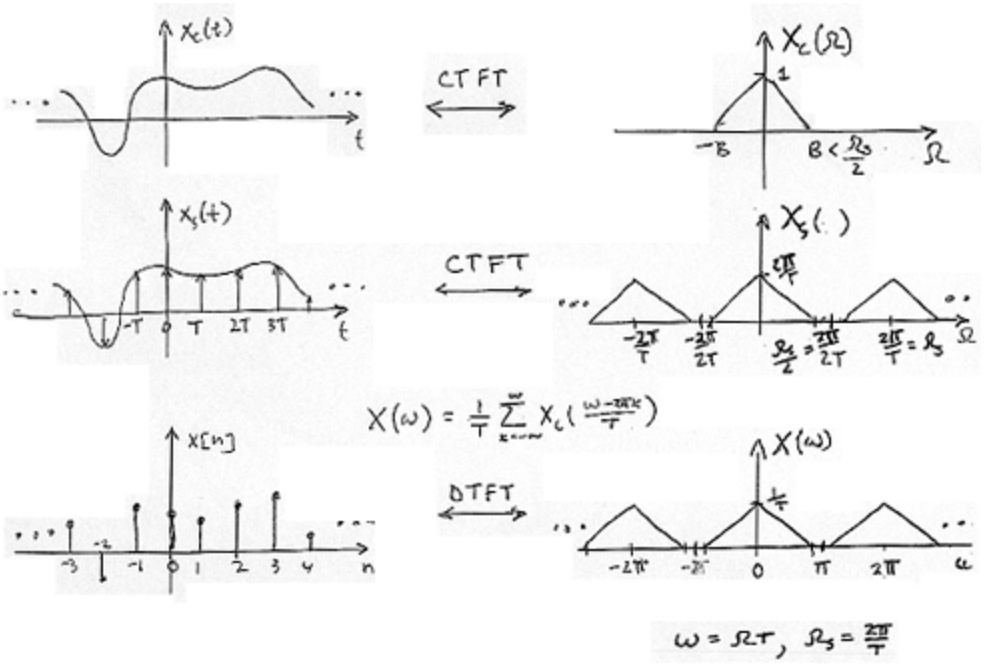
$$X_s(\Omega) = \frac{1}{T} \sum_{k=-\infty}^{\infty} X_c(\Omega - k\Omega_s)$$

Equation:

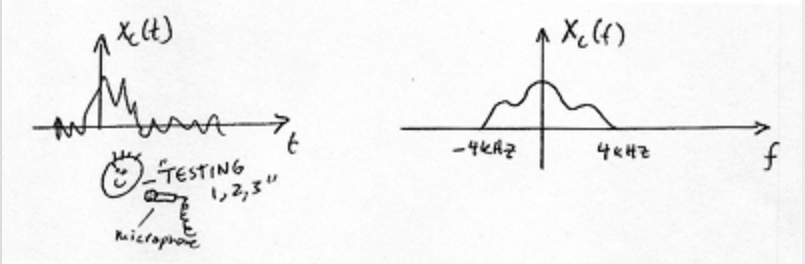
$$\begin{aligned} X(\omega) &= \frac{1}{T} \sum_{k=-\infty}^{\infty} X_c(\Omega - k\Omega_s) \\ &= \frac{1}{T} \sum_{k=-\infty}^{\infty} X_c\left(\frac{\omega - 2\pi k}{T}\right) \end{aligned}$$

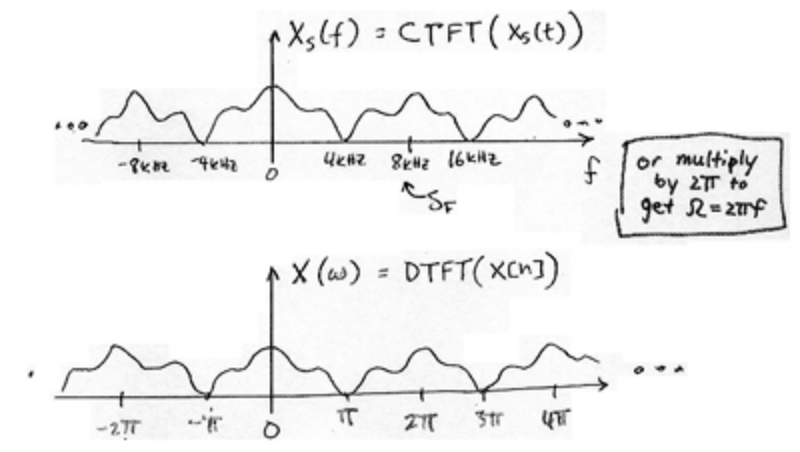
where this last part is 2π-periodic.

Sampling



Example:
Speech
Speech is intelligible if bandlimited by a CT lowpass filter to the band ±4 kHz. We can sample speech as slowly as _____?



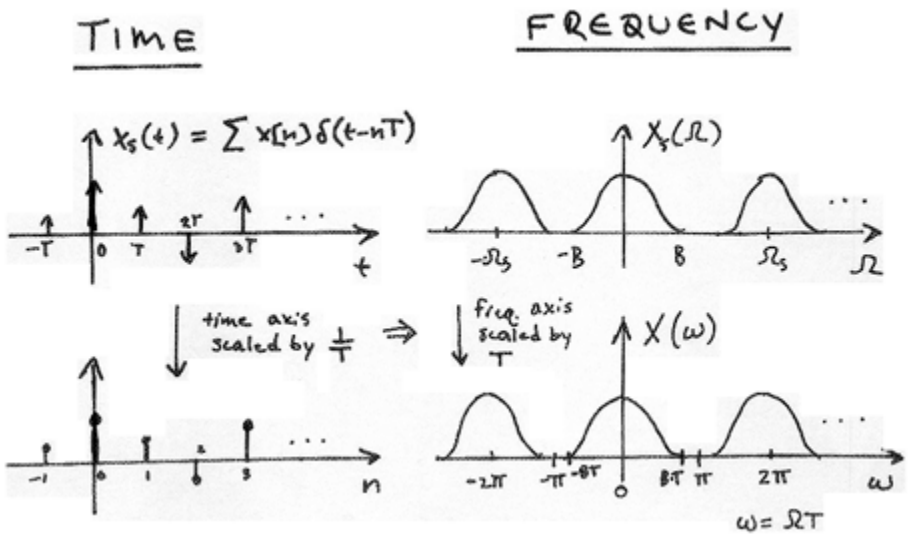


Note that there is no mention of T or Ω_s!

Relating x[n] to sampled x(t)

Recall the following equality:

$$x_s(t) = \sum_{nn} x(nT)\delta(t - nT)$$



Recall the CTFT relation:
Equation:

$$x(\alpha t) \leftrightarrow \frac{1}{\alpha} X\left(\frac{\Omega}{\alpha}\right)$$

where α is a scaling of time and $\frac{1}{\alpha}$ is a scaling in frequency.
Equation:

$$X_s(\Omega) \equiv X(\Omega T)$$

The DFT: Frequency Domain with a Computer Analysis

Introduction

We just covered ideal (and non-ideal) (time) sampling of CT signals. This enabled DT signal processing solutions for CT applications ([link]):



Much of the theoretical analysis of such systems relied on frequency domain representations. How do we carry out these frequency domain analysis on the computer? Recall the following relationships:

$$x[n] \overset{\text{DTFT}}{\longleftrightarrow} X(\omega)$$

$$x(t) \overset{\text{CTFT}}{\longleftrightarrow} X(\Omega)$$

where ω and Ω are continuous frequency variables.

Sampling DTFT

Consider the DTFT of a discrete-time (DT) signal $x[n]$. Assume $x[n]$ is of finite duration N (i.e., an N-point signal).
Equation:

$$X(\omega) = \sum_{n=0}^{N-1} x[n] e^{(-i)\omega n}$$

where $X(\omega)$ is the continuous function that is indexed by the real-valued parameter $-\pi \leq \omega \leq \pi$. The other function, $x[n]$, is a discrete function that

is indexed by integers.

We want to work with $X(\omega)$ on a computer. Why not just **sample** $X(\omega)$?
Equation:

$$\begin{aligned} X[k] &= X\left(\tfrac{2\pi}{N}k\right) \\ &= \textstyle\sum_{n=0}^{N-1} x[n]e^{(-i)2\pi\frac{k}{N}n} \end{aligned}$$

In [link] we sampled at $\omega = \frac{2\pi}{N}k$ where $k = \{0, 1, \ldots, N-1\}$ and $X[k]$ for $k = \{0, \ldots, N-1\}$ is called the **Discrete Fourier Transform (DFT)** of $x[n]$.

Example:
Finite Duration DT Signal
[missing_resource: sec8_fig2.png]
The DTFT of the image in [link] is written as follows:
Equation:

$$X(\omega) = \sum_{n=0}^{N-1} x[n]e^{(-i)\omega n}$$

where ω is any 2π-interval, for example $-\pi \leq \omega \leq \pi$.
Sample X(ω)
[missing_resource: sec8_fig3.png]
where again we sampled at $\omega = \frac{2\pi}{N}k$ where $k = \{0, 1, \ldots, M-1\}$. For example, we take

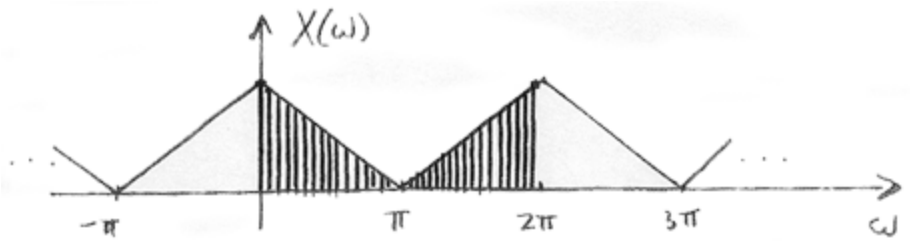
$$M = 10$$

. In the following section we will discuss in more detail how we should choose M, the number of samples in the 2π interval.
(This is precisely how we would plot $X(\omega)$ in Matlab.)

Choosing M

Case 1

Given N (length of $x[n]$), choose $M \gg N$ to obtain a dense sampling of the DTFT ([link]):



Case 2

Choose M as small as possible (to minimize the amount of computation).

In general, we require $M \geq N$ in order to represent all information in

$$\forall n, n = \{0, \ldots, N-1\} : (x[n])$$

Let's concentrate on $M = N$:

$$x[n] \overset{\text{DFT}}{\quad} X[k]$$

for $n = \{0, \ldots, N-1\}$ and $k = \{0, \ldots, N-1\}$

$$\text{numbers} \leftrightarrow \text{N numbers}$$

Discrete Fourier Transform (DFT)

Define
Equation:

$$X[k] \equiv X\left(\frac{2\pi k}{N}\right)$$

where $N = \text{length}\left(x[n]\right)$ and $k = \{0, \ldots, N-1\}$. In this case, $M = N$.
Equation:

DFT

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{(-i)2\pi \frac{k}{N} n}$$

Equation:

Inverse DFT (IDFT)

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] e^{i2\pi \frac{k}{N} n}$$

Interpretation

Represent $x[n]$ in terms of a sum of N complex sinusoids of amplitudes $X[k]$ and frequencies

$$\forall k, k \in \{0, \ldots, N-1\} : \left(\omega_k = \frac{2\pi k}{N}\right)$$

Note: Fourier Series with fundamental frequency $\frac{2\pi}{N}$

Remark 1

IDFT treats $x[n]$ as though it were N-periodic.

Equation:

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] e^{i 2\pi \frac{k}{N} n}$$

where $n \in \{0, \ldots, N-1\}$

Exercise:

Problem: What about other values of n?

Solution:

$$x[n+N] = ???$$

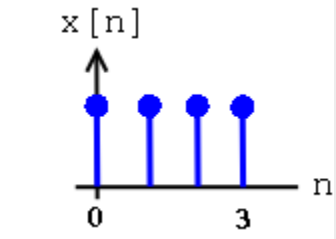
Remark 2

Proof that the IDFT inverts the DFT for $n \in \{0, \ldots, N-1\}$

Equation:

$$\begin{aligned} \frac{1}{N} \sum_{k=0}^{N-1} X[k] e^{i 2\pi \frac{k}{N} n} &= \frac{1}{N} \sum_{k=0}^{N-1} \sum_{m=0}^{N-1} x[m] e^{(-i) 2\pi \frac{k}{N} m} e^{i 2\pi \frac{k}{N} n} \\ &= ??? \end{aligned}$$

Example:
Computing DFT
Given the following discrete-time signal ([link]) with $N = 4$, we will compute the DFT using two different methods (the DFT Formula and Sample DTFT):



1. DFT Formula
   Equation:

$$\begin{aligned} X[k] &= \sum_{n=0}^{N-1} x[n] e^{(-i)2\pi \frac{k}{N} n} \\ &= 1 + e^{(-i)2\pi \frac{k}{4}} + e^{(-i)2\pi \frac{k}{4} 2} + e^{(-i)2\pi \frac{k}{4} 3} \\ &= 1 + e^{(-i)\frac{\pi}{2} k} + e^{(-i)\pi k} + e^{(-i)\frac{3}{2}\pi k} \end{aligned}$$

   Using the above equation, we can solve and get the following results:

$$x[0] = 4$$

$$x[1] = 0$$

$$x[2] = 0$$

$$x[3] = 0$$

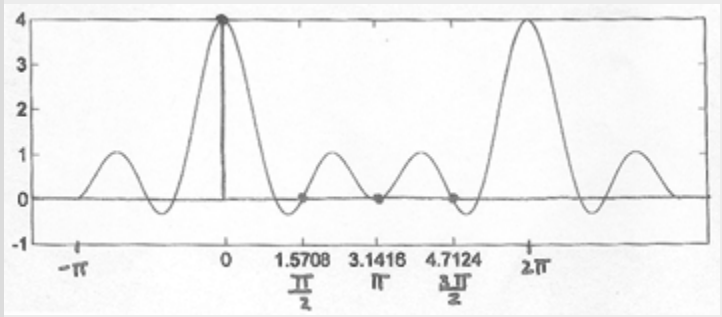
2. Sample DTFT. Using the same figure, [link], we will take the DTFT of the signal and get the following equations:
   Equation:

$$\begin{aligned} X(\omega) &= \sum_{n=0}^{3} e^{(-i)\omega n} \\ &= \frac{1 - e^{(-i)4\omega}}{1 - e^{(-i)\omega}} \\ &= ??? \end{aligned}$$

   Our sample points will be:

$$\omega_k = \frac{2\pi k}{4} = \frac{\pi}{2} k$$

where $k = \{0, 1, 2, 3\}$ ([link]).

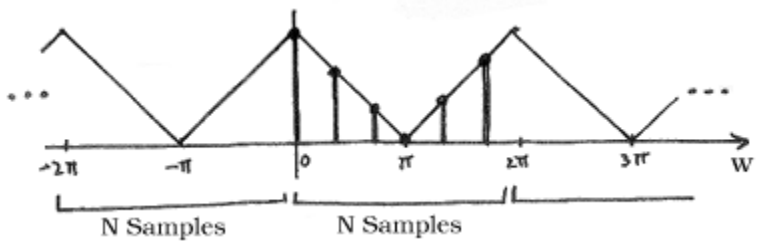


Periodicity of the DFT

DFT $X[k]$ consists of **samples** of DTFT, so $X(\omega)$, a 2π-periodic DTFT signal, can be converted to $X[k]$, an N-periodic DFT.
Equation:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{(-i)2\pi \frac{k}{N} n}$$

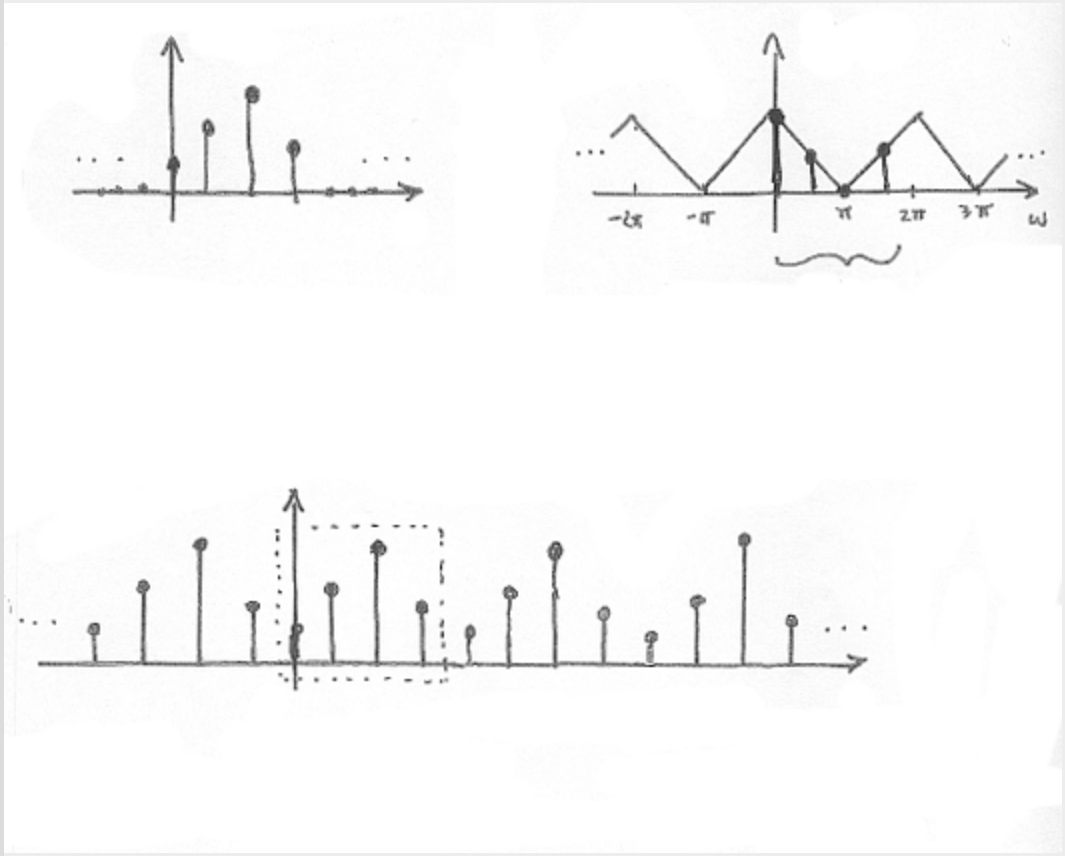
where $e^{(-i)2\pi \frac{k}{N} n}$ is an N-periodic basis function (See [link]).



Also, recall,
Equation:

$$
\begin{aligned}
x[n] &= \tfrac{1}{N}\sum_{n=0}^{N-1} X[k] e^{i2\pi\frac{k}{N}n} \\
&= \tfrac{1}{N}\sum_{n=0}^{N-1} X[k] e^{i2\pi\frac{k}{N}(n+mN)} \\
&= ???
\end{aligned}
$$

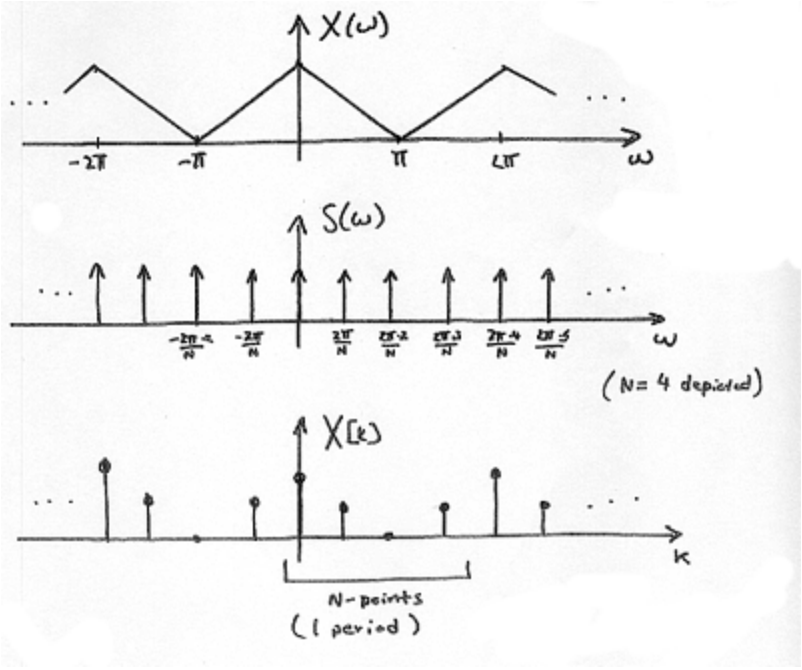
Example:
Illustration



Note: When we deal with the DFT, we need to remember that, in effect, this treats the signal as an N-periodic sequence.

A Sampling Perspective

Think of sampling the continuous function $X(\omega)$, as depicted in [link]. $S(\omega)$ will represent the sampling function applied to $X(\omega)$ and is illustrated in [link] as well. This will result in our discrete-time sequence, $X[k]$.



Note: Remember the multiplication in the frequency domain is equal to convolution in the time domain!

Inverse DTFT of S(ω)

Equation:

$$\sum_{k=-\infty}^{\infty} \delta\left(\omega - \frac{2\pi k}{N}\right)$$

Given the above equation, we can take the DTFT and get the following equation:
Equation:

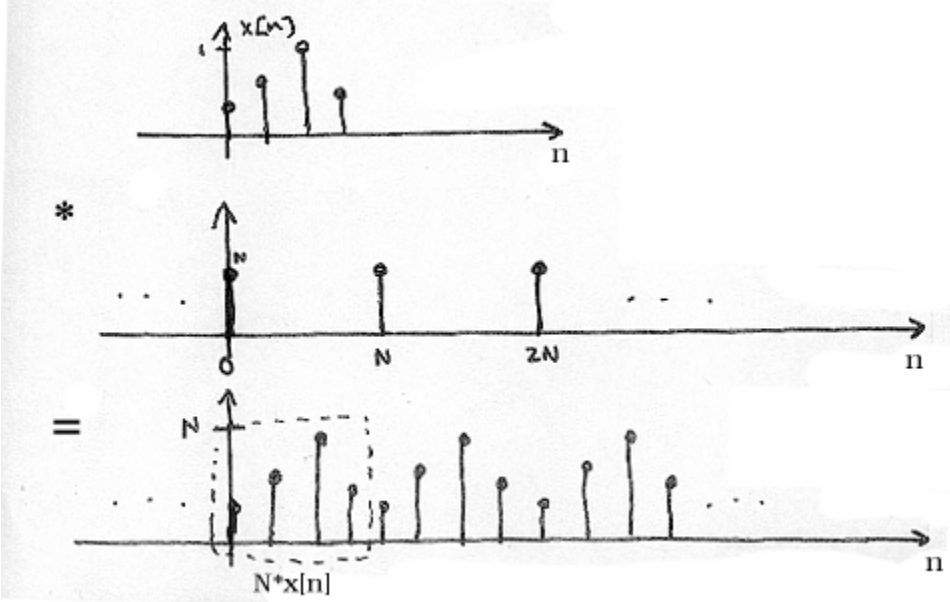
$$N \sum_{m=-\infty}^{\infty} \delta[n - mN] \equiv S[n]$$

Exercise:

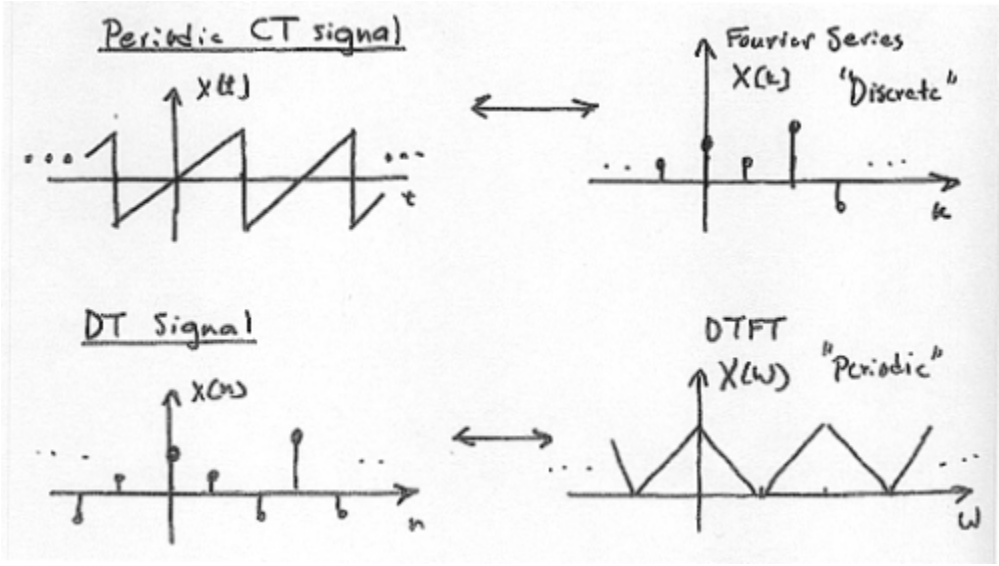
> **Problem:** Why does [link] equal $S[n]$?

> **Solution:**
>
> $S[n]$ is N-periodic, so it has the following Fourier Series:
> **Equation:**
>
> $$\begin{aligned} c_k &= \frac{1}{N} \int_{-\frac{N}{2}}^{\frac{N}{2}} \delta[n] e^{(-i)2\pi \frac{k}{N} n} \, \mathrm{d}\, n \\ &= \frac{1}{N} \end{aligned}$$
>
> **Equation:**
>
> $$S[n] = \sum_{k=-\infty}^{\infty} e^{(-i)2\pi \frac{k}{N} n}$$
>
> where the DTFT of the exponential in the above equation is equal to $\delta\left(\omega - \frac{2\pi k}{N}\right)$.

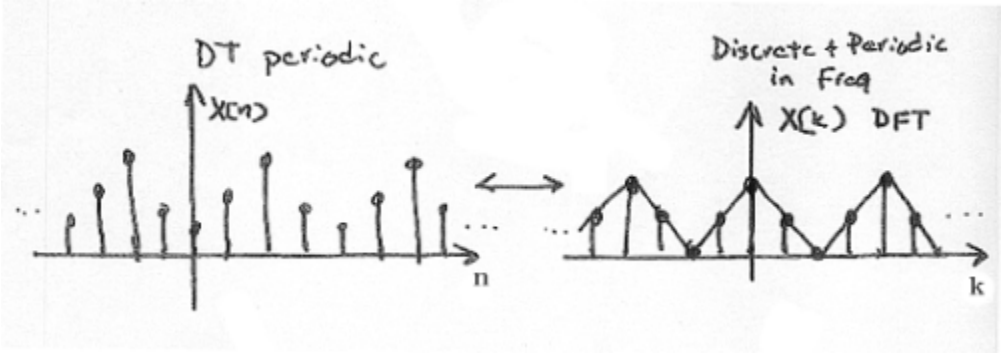
So, in the time-domain we have ([link]):



Connections



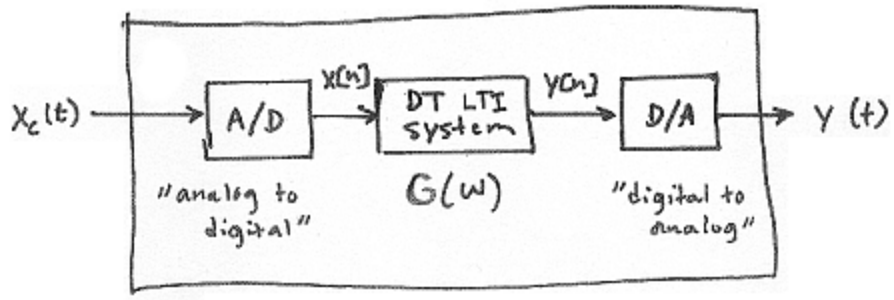
Combine signals in [link] to get signals in [link].


DT periodic
X[n]
n
Discrete & Periodic
in Freq
X(k) DFT
k

Discrete-Time Processing of CT Signals

DT Processing of CT Signals

DSP System



Analysis

Equation:

$$Y_c(\Omega) = H_{\text{LP}}(\Omega)Y(\Omega T)$$

where we know that $Y(\omega) = X(\omega)G(\omega)$ and $G(\omega)$ is the frequency response of the DT LTI system. Also, remember that

$$\omega \equiv \Omega T$$

So,
Equation:

$$Y_c(\Omega) = H_{\text{LP}}(\Omega)G(\Omega T)X(\Omega T)$$

where $Y_c(\Omega)$ and $H_{\text{LP}}(\Omega)$ are CTFTs and $G(\Omega T)$ and $X(\Omega T)$ are DTFTs.

Note:

$$X(\omega) = \frac{2\pi}{T} \sum_{k=-\infty}^{\infty} X_c\left(\frac{\omega - 2\pi k}{T}\right)$$

OR

$$X(\Omega T) = \frac{2\pi}{T} \sum_{k=-\infty}^{\infty} X_c(\Omega - k\Omega_s)$$

Therefore our final output signal, $Y_c(\Omega)$, will be:
Equation:

$$Y_c(\Omega) = H_{\text{LP}}(\Omega) G(\Omega T) \left(\frac{2\pi}{T} \sum_{k=-\infty}^{\infty} X_c(\Omega - k\Omega_s) \right)$$

Now, if $X_c(\Omega)$ is bandlimited to $\left[-\frac{\Omega_s}{2}, \frac{\Omega_s}{2}\right]$ and we use the usual lowpass reconstruction filter in the D/A, [link]:

[missing_resource: sec9_fig2.png]

Then,
Equation:

$$Y_c(\Omega) = \begin{cases} G(\Omega T) X_c(\Omega) & \text{if} \quad |\Omega| < \frac{\Omega_s}{2} \\ 0 \quad \text{otherwise} \end{cases}$$

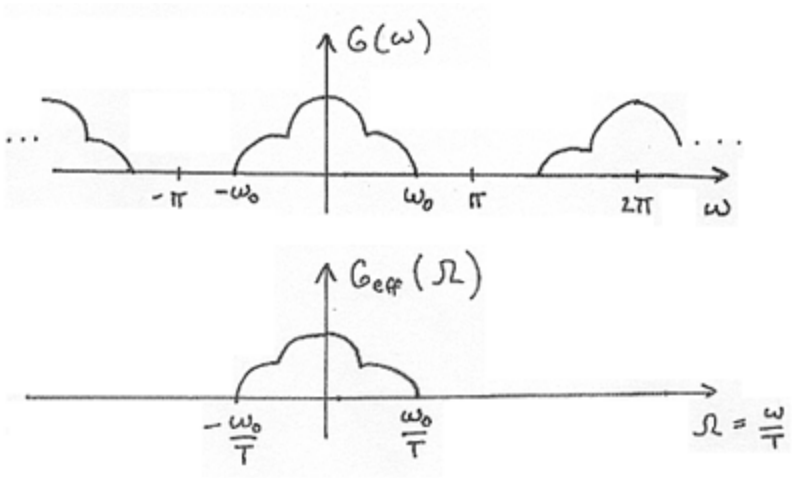
Summary

For bandlimited signals sampled at or above the Nyquist rate, we can relate the input and output of the DSP system by:
Equation:

$$Y_c(\Omega) = G_{\text{eff}}(\Omega) X_c(\Omega)$$

where

$$G_{\text{eff}}(\Omega) = \begin{cases} G(\Omega T) \ \text{ if } \ |\Omega| < \frac{\Omega_s}{2} \\ 0 \ \text{ otherwise} \end{cases}$$



Note

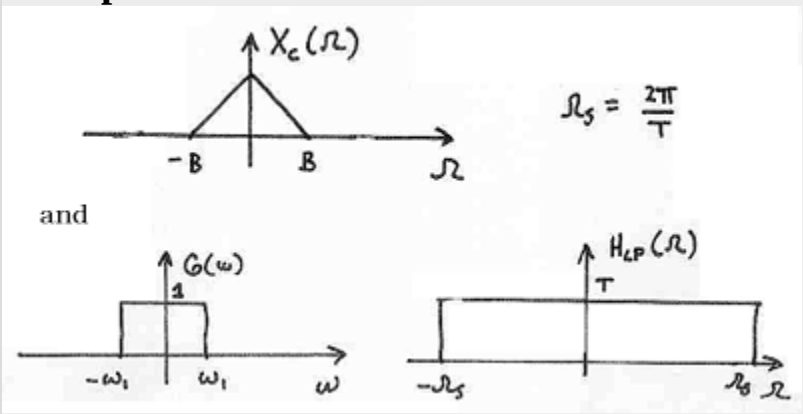
$G_{\text{eff}}(\Omega)$ is LTI if and only if the following two conditions are satisfied:

1. $G(\omega)$ is LTI (in DT).
2. $X_c(T)$ is bandlimited and sampling rate equal to or greater than Nyquist. For example, if we had a simple pulse described by

$$X_c(t) = u(t - T_0) - u(t - T_1)$$

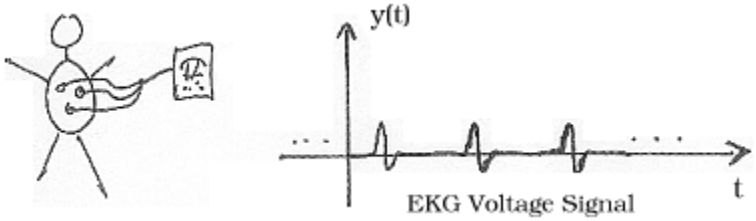
where $T_1 > T_0$. If the sampling period $T > T_1 - T_0$, then some samples might "miss" the pulse while others might not be "missed." This is what we term **time-varying behavior**.

Example:

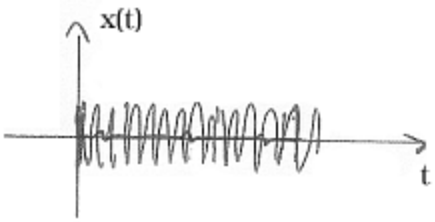


If $\frac{2\pi}{T} > 2B$ and $\omega_1 < BT$, determine and sketch $Y_c(\Omega)$ using [link].

Application: 60Hz Noise Removal

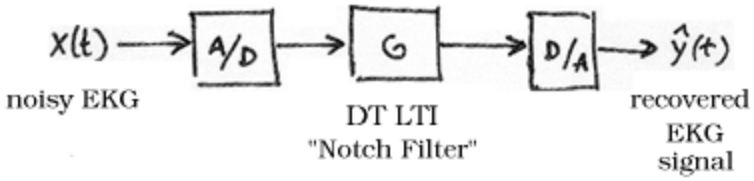


Unfortunately, in real-world situations electrodes also pick up ambient 60 Hz signals from lights, computers, etc.. In fact, usually this "60 Hz noise" is much greater in amplitude than the EKG signal shown in [link]. [link] shows the EKG signal; it is barely noticeable as it has become overwhelmed by noise.



Our EKG signal, $y(t)$, is overwhelmed by noise.

DSP Solution



[missing_resource: sec9_fig7b.png]

Sampling Period/Rate

First we must note that $|Y(\Omega)|$ is **bandlimited** to ±60 Hz. Therefore, the minimum rate should be 120 Hz. In order to get the best results we should set

$$f_s = 240\,\text{Hz}$$

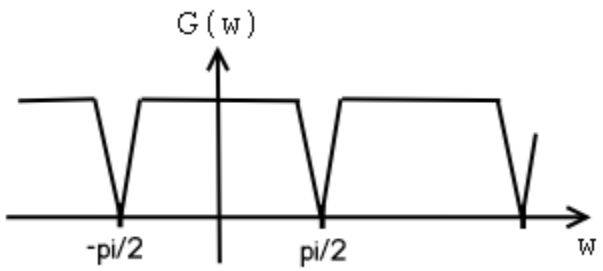
.

$$\Omega_s = 2\pi \times \left(240 \frac{\text{rad}}{s} \right)$$

[missing_resource: sec9_fig8.png]

Digital Filter

Therefore, we want to design a digital filter that will remove the 60Hz component and preserve the rest.



Difference Equation
(Blank Abstract)

Introduction

One of the most important concepts of DSP is to be able to properly represent the input/output relationship to a given LTI system. A linear constant-coefficient **difference equation** (LCCDE) serves as a way to express just this relationship in a discrete-time system. Writing the sequence of inputs and outputs, which represent the characteristics of the LTI system, as a difference equation help in understanding and manipulating a system.

difference equation
: An equation that shows the relationship between consecutive values of a sequence and the differences among them. They are often rearranged as a recursive formula so that a systems output can be computed from the input signal and past outputs.

Example:
Equation:

$$y[n] + 7y[n-1] + 2y[n-2] = x[n] - 4x[n-1]$$

General Formulas for the Difference Equation

As stated briefly in the definition above, a difference equation is a very useful tool in describing and calculating the output of the system described by the formula for a given sample n. The key property of the difference equation is its ability to help easily find the transform, $H(z)$, of a system. In the following two subsections, we will look at the general form of the difference equation and the general conversion to a z-transform directly from the difference equation.

Difference Equation

The general form of a linear, constant-coefficient difference equation (LCCDE), is shown below:
Equation:

$$\sum_{k=0}^{N} a_k y[n-k] = \sum_{k=0}^{M} b_k x[n-k]$$

We can also write the general form to easily express a recursive output, which looks like this:
Equation:

$$y[n] = -\sum_{k=1}^{N} a_k y[n-k] + \sum_{k=0}^{M} b_k x[n-k]$$

From this equation, note that $y[n-k]$ represents the outputs and $x[n-k]$ represents the inputs. The value of N represents the **order** of the difference equation and corresponds to the memory of the system being represented. Because this equation relies on past values of the output, in order to compute a numerical solution, certain past outputs, referred to as the **initial conditions**, must be known.

Conversion to Z-Transform

Using the above formula, [link], we can easily generalize the **transfer function**, $H(z)$, for any difference equation. Below are the steps taken to convert any difference equation into its transfer function, i.e. z-transform. The first step involves taking the Fourier Transform of all the terms in [link]. Then we use the linearity property to pull the transform inside the summation and the time-shifting property of the z-transform to change the time-shifting terms to exponentials. Once this is done, we arrive at the following equation: $a_0 = 1$.
Equation:

$$Y(z) = -\sum_{k=1}^{N} a_k Y(z) z^{-k} + \sum_{k=0}^{M} b_k X(z) z^{-k}$$

Equation:

$$\begin{aligned} H(z) &= \frac{Y(z)}{X(z)} \\ &= \frac{\sum_{k=0}^{M} b_k z^{-k}}{1+\sum_{k=1}^{N} a_k z^{-k}} \end{aligned}$$

Conversion to Frequency Response

Once the z-transform has been calculated from the difference equation, we can go one step further to define the frequency response of the system, or filter, that is being represented by the difference equation.

Note: Remember that the reason we are dealing with these formulas is to be able to aid us in filter design. A LCCDE is one of the easiest ways to represent FIR filters. By being able to find the frequency response, we will be able to look at the basic properties of any filter represented by a simple LCCDE.

Below is the general formula for the frequency response of a z-transform. The conversion is simple a matter of taking the z-transform formula, $H(z)$, and replacing every instance of z with e^{iw}.
Equation:

$$\begin{aligned} H(w) &= H(z)|_{z,z=e^{iw}} \\ &= \frac{\sum_{k=0}^{M} b_k e^{-(iwk)}}{\sum_{k=0}^{N} a_k e^{-(iwk)}} \end{aligned}$$

Once you understand the derivation of this formula, look at the module concerning Filter Design from the Z-Transform for a look into how all of these ideas of the Z-transform, Difference Equation, and Pole/Zero Plots play a role in filter design.

Example

Example:
Finding Difference Equation
Below is a basic example showing the opposite of the steps above: given a transfer function one can easily calculate the systems difference equation.
Equation:

$$H(z) = \frac{(z+1)^2}{\left(z-\frac{1}{2}\right)\left(z+\frac{3}{4}\right)}$$

Given this transfer function of a time-domain filter, we want to find the difference equation. To begin with, expand both polynomials and divide them by the highest order z.
Equation:

$$\begin{aligned} H(z) &= \frac{(z+1)(z+1)}{\left(z-\frac{1}{2}\right)\left(z+\frac{3}{4}\right)} \\ &= \frac{z^2+2z+1}{z^2+2z+1-\frac{3}{8}} \\ &= \frac{1+2z^{-1}+z^{-2}}{1+\frac{1}{4}z^{-1}-\frac{3}{8}z^{-2}} \end{aligned}$$

From this transfer function, the coefficients of the two polynomials will be our a_k and b_k values found in the general difference equation formula, [link]. Using these coefficients and the above form of the transfer function, we can easily write the difference equation:
Equation:

$$x[n] + 2x[n-1] + x[n-2] = y[n] + \frac{1}{4}y[n-1] - \frac{3}{8}y[n-2]$$

In our final step, we can rewrite the difference equation in its more common form showing the recursive nature of the system.
Equation:

$$y[n] = x[n] + 2x[n-1] + x[n-2] + \frac{-1}{4}y[n-1] + \frac{3}{8}y[n-2]$$

Solving a LCCDE

In order for a linear constant-coefficient difference equation to be useful in analyzing a LTI system, we must be able to find the systems output based upon a known input, $x(n)$, and a set of initial conditions. Two common methods exist for solving a LCCDE: the **direct method** and the **indirect method**, the later being based on the z-transform. Below we will briefly discuss the formulas for solving a LCCDE using each of these methods.

Direct Method

The final solution to the output based on the direct method is the sum of two parts, expressed in the following equation:
Equation:

$$y(n) = y_h(n) + y_p(n)$$

The first part, $y_h(n)$, is referred to as the **homogeneous solution** and the second part, $y_h(n)$, is referred to as **particular solution**. The following method is very similar to that used to solve many differential equations, so if you have taken a differential calculus course or used differential equations before then this should seem very familiar.

Homogeneous Solution

We begin by assuming that the input is zero, $x(n) = 0$. Now we simply need to solve the homogeneous difference equation:
Equation:

$$\sum_{k=0}^{N} a_k y[n-k] = 0$$

In order to solve this, we will make the assumption that the solution is in the form of an exponential. We will use lambda, λ, to represent our exponential terms. We now have to solve the following equation:
Equation:

$$\sum_{k=0}^{N} a_k \lambda^{n-k} = 0$$

We can expand this equation out and factor out all of the lambda terms. This will give us a large polynomial in parenthesis, which is referred to as the **characteristic polynomial**. The roots of this polynomial will be the key to solving the homogeneous equation. If there are all distinct roots, then the general solution to the equation will be as follows:
Equation:

$$y_h(n) = C_1(\lambda_1)^n + C_2(\lambda_2)^n + \ldots + C_N(\lambda_N)^n$$

However, if the characteristic equation contains multiple roots then the above general solution will be slightly different. Below we have the modified version for an equation where λ_1 has K multiple roots:
Equation:

$$y_h(n) = C_1(\lambda_1)^n + C_1 n(\lambda_1)^n + C_1 n^2(\lambda_1)^n + \ldots + C_1 n^{K-1}(\lambda_1)^n + C_2(\lambda_2)^n + \ldots + C_N(\lambda_N)^n$$

Particular Solution

The particular solution, $y_p(n)$, will be any solution that will solve the general difference equation:
Equation:

$$\sum_{k=0}^{N} a_k y_p(n-k) = \sum_{k=0}^{M} b_k x(n-k)$$

In order to solve, our guess for the solution to $y_p(n)$ will take on the form of the input, $x(n)$. After guessing at a solution to the above equation involving the particular solution, one only needs to plug the solution into the difference equation and solve it out.

Indirect Method

The indirect method utilizes the relationship between the difference equation and z-transform, discussed earlier, to find a solution. The basic idea is to convert the difference equation into a z-transform, as described above, to get the resulting output, $Y(z)$. Then by inverse transforming this and using partial-fraction expansion, we can arrive at the solution.
Equation:

$$Z\{y\,(n+1) - y\,(n)\} = zY\,(z) - y\,(0)$$

This can be interatively extended to an arbitrary order derivative as in Equation [link].
Equation:

$$Z\left\{ -\sum_{m=0}^{N-1} y\,(n\text{-}m) \right\} = z^n Y\,(z) - \sum_{m=0}^{N-1} z^{n-m-1} y^{(m)}\,(0)$$

Now, the Laplace transform of each side of the differential equation can be taken
Equation:

$$Z\left\{\sum_{k=0}^{N} a_k \left[y\left(n\text{-}m\text{+}1\right) - \sum_{m=0}^{N-1} y\left(n\text{-}m\right) y\left(n\right) \right] = Z\{x(n)\} \right\}$$

which by linearity results in
Equation:

$$\sum_{k=0}^{N} a_k Z\left\{ y\left(n\text{-}m\text{+}1\right) - \sum_{m=0}^{N-1} y\left(n\text{-}m\right) y\left(n\right) \right\} = Z\{x(n)\}$$

and by differentiation properties in
Equation:

$$\sum_{k=0}^{N} a_k \left(z^k Z\{y(n)\} - \sum_{m=0}^{N-1} z^{k-m-1} y^{(m)}\left(0\right) \right) = Z\{x(n)\}.$$

Rearranging terms to isolate the Laplace transform of the output,
Equation:

$$Z\{y(n)\} = \frac{Z\{x(n)\} + \sum_{k=0}^{N} \sum_{m=0}^{k-1} a_k z^{k-m-1} y^{(m)}\left(0\right)}{\sum_{k=0}^{N} a_k z^k}.$$

Thus, it is found that
Equation:

$$Y\left(z\right) = \frac{X\left(z\right) + \sum_{k=0}^{N} \sum_{m=0}^{k-1} a_k z^{k-m-1} y^{(m)}\left(0\right)}{\sum_{k=0}^{N} a_k z^k}.$$

In order to find the output, it only remains to find the Laplace transform $X(z)$ of the input, substitute the initial conditions, and compute the inverse Z-transform of the result. Partial fraction expansions are often required for this last step. This may sound daunting while looking at [link], but it is often easy in practice, especially for low order difference equations. [link] can also be used to determine the transfer function and frequency response.

As an example, consider the difference equation
Equation:

$$y\left[n\text{-}2\right] + 4y\left[n\text{-}1\right] + 3y\left[n\right] = \cos\left(n\right)$$

with the initial conditions $y'(0) = 1$ and $y(0) = 0$ Using the method described above, the Z transform of the solution $y[n]$ is given by
Equation:

$$Y[z] = \frac{z}{[z^2+1][z+1][z+3]} + \frac{1}{[z+1][z+3]}.$$

Performing a partial fraction decomposition, this also equals
Equation:

$$Y[z] = .25\frac{1}{z+1} - .35\frac{1}{z+3} + .1\frac{z}{z^2+1} + .2\frac{1}{z^2+1}.$$

Computing the inverse Laplace transform,
Equation:

$$y(n) = \left(.25z^{-n} - .35z^{-3n} + .1\cos(n) + .2\sin(n)\right)u(n).$$

One can check that this satisfies that this satisfies both the differential equation and the initial conditions.

The Z Transform: Definition
A brief definition of the z-transform, explaining its relationship with the Fourier transform and its region of convergence, ROC.

Basic Definition of the Z-Transform

The **z-transform** of a sequence is defined as
Equation:

$$X(z) = \sum_{n=-\infty}^{\infty} x[n] z^{-n}$$

Sometimes this equation is referred to as the **bilateral z-transform**. At times the z-transform is defined as
Equation:

$$X(z) = \sum_{n=0}^{\infty} x[n] z^{-n}$$

which is known as the **unilateral z-transform**.

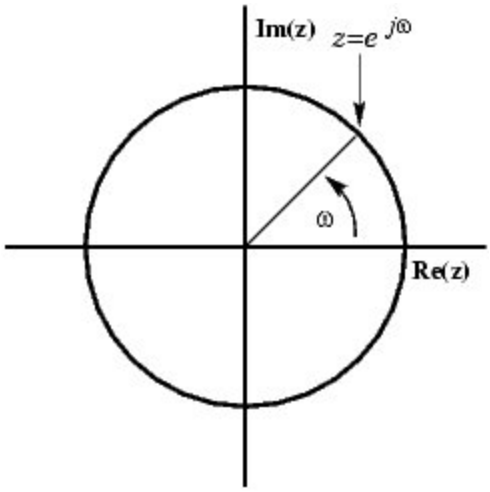
There is a close relationship between the z-transform and the **Fourier transform** of a discrete time signal, which is defined as
Equation:

$$X\left(e^{i\omega}\right) = \sum_{n=-\infty}^{\infty} x[n] e^{-(i\omega n)}$$

Notice that that when the z^{-n} is replaced with $e^{-(i\omega n)}$ the z-transform reduces to the Fourier Transform. When the Fourier Transform exists, $z = e^{i\omega}$, which is to have the magnitude of z equal to unity.

The Complex Plane

In order to get further insight into the relationship between the Fourier Transform and the Z-Transform it is useful to look at the complex plane or **z-plane**. Take a look at the complex plane:
Z-Plane



The Z-plane is a complex plane with an imaginary and real axis referring to the complex-valued variable z. The position on the complex plane is given by $re^{i\omega}$, and the angle from the positive, real axis around the plane is denoted by ω. $X(z)$ is defined everywhere on this plane. $X\left(e^{i\omega}\right)$ on the other hand is defined only where $|z| = 1$, which is referred to as the unit circle. So for example, $\omega = 1$ at $z = 1$ and $\omega = \pi$ at $z = -1$. This is useful because, by representing the Fourier transform as the z-transform on the unit circle, the periodicity of Fourier transform is easily seen.

Region of Convergence

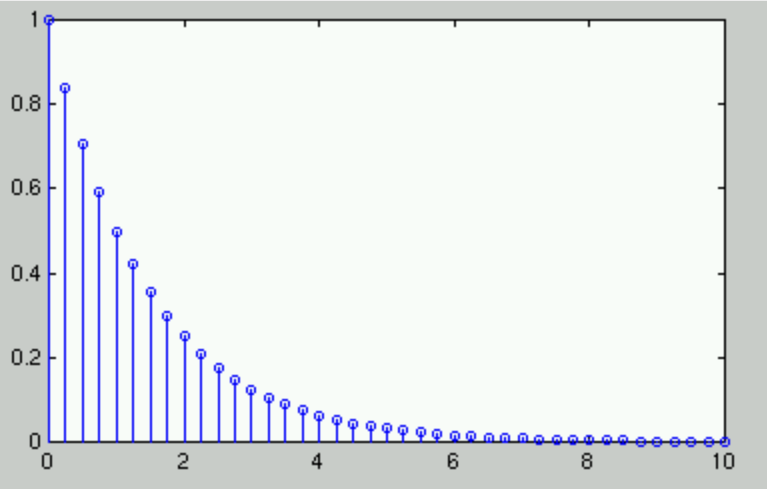
The region of convergence, known as the **ROC**, is important to understand because it defines the region where the z-transform exists. The ROC for a given $x[n]$, is defined as the range of z for which the z-transform converges. Since the z-transform is a **power series**, it converges when $x[n]z^{-n}$ is absolutely summable. Stated differently,
Equation:

$$\sum_{n=-\infty}^{\infty} \left|x[n]z^{-n}\right| < \infty$$

must be satisfied for convergence. This is best illustrated by looking at the different ROC's of the z-transforms of $\alpha^n u[n]$ and $\alpha^n u[n-1]$.

Example:
For
Equation:

$$x[n] = \alpha^n u[n]$$



$x[n] = \alpha^n u[n]$ where $\alpha = 0.5$.

Equation:

$$\begin{aligned} X(z) &= \sum_{n=-\infty}^{\infty} x[n] z^{-n} \\ &= \sum_{n=-\infty}^{\infty} \alpha^n u[n] z^{-n} \\ &= \sum_{n=0}^{\infty} \alpha^n z^{-n} \\ &= \sum_{n=0}^{\infty} \left(\alpha z^{-1}\right)^n \end{aligned}$$

This sequence is an example of a right-sided exponential sequence because it is nonzero for $n \geq 0$. It only converges when $\left|\alpha z^{-1}\right| < 1$. When it converges,

Equation:

$$\begin{aligned} X(z) &= \frac{1}{1-\alpha z^{-1}} \\ &= \frac{z}{z-\alpha} \end{aligned}$$

If $\left|\alpha z^{-1}\right| \geq 1$, then the series, $\sum_{n=0}^{\infty} \left(\alpha z^{-1}\right)^n$ does not converge. Thus the ROC is the range of values where

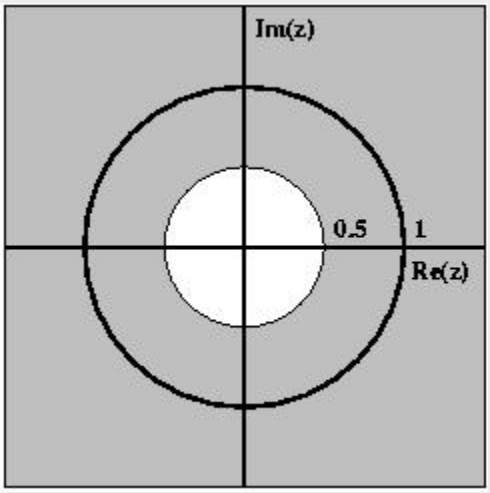
Equation:

$$\left|\alpha z^{-1}\right| < 1$$

or, equivalently,

Equation:

$$|z| > |\alpha|$$



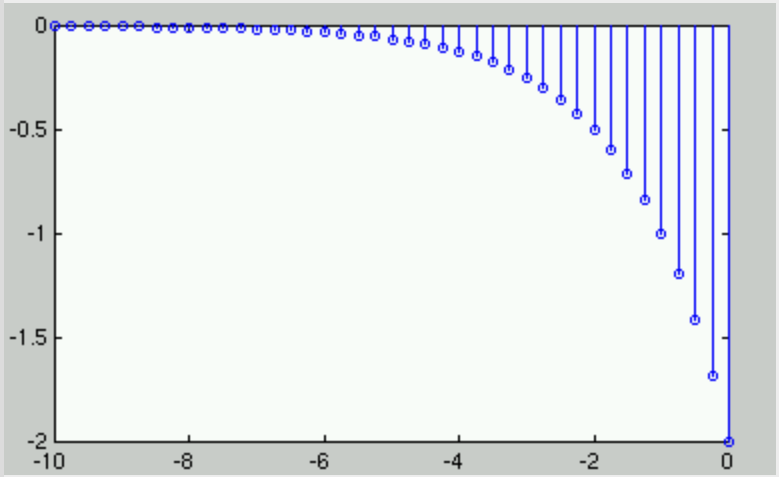
ROC for $x[n] = \alpha^n u[n]$
where $\alpha = 0.5$

Example:

For

Equation:

$$x[n] = (-\alpha^n)u[(-n) - 1]$$



$x[n] = (-\alpha^n)u[(-n) - 1]$ where
$\alpha = 0.5.$

Equation:

$$\begin{aligned} X(z) &= \sum_{n=-\infty}^{\infty} x[n]z^{-n} \\ &= \sum_{n=-\infty}^{\infty} (-\alpha^n)u[\text{-n} - 1]z^{-n} \\ &= -\sum_{n=-\infty}^{-1} \alpha^n z^{-n} \\ &= -\sum_{n=-\infty}^{-1} \left(\alpha^{-1}z\right)^{-n} \\ &= -\sum_{n=1}^{\infty} \left(\alpha^{-1}z\right)^{n} \\ &= 1 - \sum_{n=0}^{\infty} \left(\alpha^{-1}z\right)^{n} \end{aligned}$$

The ROC in this case is the range of values where
Equation:

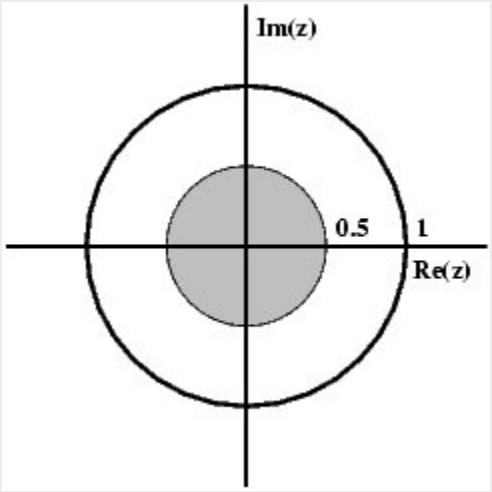
$$\left|\alpha^{-1}z\right| < 1$$

or, equivalently,
Equation:

$$|z| < |\alpha|$$

If the ROC is satisfied, then
Equation:

$$\begin{aligned} X(z) &= 1 - \frac{1}{1-\alpha^{-1}z} \\ &= \frac{z}{z-\alpha} \end{aligned}$$



ROC for
$x[n] = (-\alpha^n)u[(-n) - 1]$

Table of Common z-Transforms
Lists the z-transform and region of convergence (ROC) for several common discrete-time signals.

The table below provides a number of unilateral and bilateral **z-transforms**. The table also specifies the region of convergence.

Note: The notation for z found in the table below may differ from that found in other tables. For example, the basic z-transform of $u[n]$ can be written as either of the following two expressions, which are equivalent:

Equation:

$$\frac{z}{z-1} = \frac{1}{1-z^{-1}}$$

Signal	Z-Transform	ROC
$\delta[n-k]$	z^{-k}	$\mathrm{All}(z)$
$u[n]$	$\frac{z}{z-1}$	$\lvert z\rvert > 1$
$-u[(-n)-1]$	$\frac{z}{z-1}$	$\lvert z\rvert < 1$
$nu[n]$	$\frac{z}{(z-1)^2}$	$\lvert z\rvert > 1$

Signal	Z-Transform	ROC
$n^2u[n]$	$\frac{z(z+1)}{(z-1)^3}$	$\|z\| > 1$
$n^3u[n]$	$\frac{z\left(z^2+4z+1\right)}{(z-1)^4}$	$\|z\| > 1$
$(-\alpha^n)u[(-n)-1]$	$\frac{\mathrm{z}}{z-\alpha}$	$\|z\| < \|\alpha\|$
$\alpha^n u[n]$	$\frac{z}{z-\alpha}$	$\|z\| > \|\alpha\|$
$n\alpha^n u[n]$	$\frac{\alpha z}{(z-\alpha)^2}$	$\|z\| > \|\alpha\|$
$n^2\alpha^n u[n]$	$\frac{\alpha z(z+\alpha)}{(z-\alpha)^3}$	$\|z\| > \|\alpha\|$
$\frac{\prod_{k\ 1}^{m} n-k+1}{\alpha^m m!}\alpha^n u[n]$	$\frac{z}{(z-\alpha)^{m+1}}$	
$\gamma^n \cos(\alpha n)u[n]$	$\frac{z(z-\gamma\cos(\alpha))}{z^2-(2\gamma\cos(\alpha))z+\gamma^2}$	$\|z\| > \|\gamma\|$
$\gamma^n \sin(\alpha n)u[n]$	$\frac{z\gamma\sin(\alpha)}{z^2-(2\gamma\cos(\alpha))z+\gamma^2}$	$\|z\| > \|\gamma\|$

Understanding Pole/Zero Plots on the Z-Plane
This module will look at the relationships between the z-transform and the complex plane. Specifically, the creation of pole/zero plots and some of their useful properties are discussed.

Introduction to Poles and Zeros of the Z-Transform

It is quite difficult to qualitatively analyze the Laplace transform and Z-transform, since mappings of their magnitude and phase or real part and imaginary part result in multiple mappings of 2-dimensional surfaces in 3-dimensional space. For this reason, it is very common to examine a plot of a transfer function's poles and zeros to try to gain a qualitative idea of what a system does.

Once the Z-transform of a system has been determined, one can use the information contained in function's polynomials to graphically represent the function and easily observe many defining characteristics. The Z-transform will have the below structure, based on Rational Functions:
Equation:

$$X(z) = \frac{P(z)}{Q(z)}$$

The two polynomials, $P(z)$ and $Q(z)$, allow us to find the poles and zeros of the Z-Transform.

zeros
: The value(s) for z where $P(z) = 0$.
: The complex frequencies that make the overall gain of the filter transfer function zero.

poles
: The value(s) for z where $Q(z) = 0$.
: The complex frequencies that make the overall gain of the filter transfer function infinite.

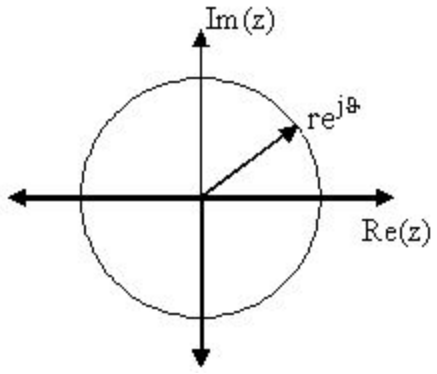
Example:
Below is a simple transfer function with the poles and zeros shown below it.

$$H(z) = \frac{z+1}{\left(z-\frac{1}{2}\right)\left(z+\frac{3}{4}\right)}$$

The zeros are: $\{-1\}$
The poles are: $\left\{\frac{1}{2}, -\frac{3}{4}\right\}$

The Z-Plane

Once the poles and zeros have been found for a given Z-Transform, they can be plotted onto the Z-Plane. The Z-plane is a complex plane with an imaginary and real axis referring to the complex-valued variable z. The position on the complex plane is given by $re^{i\theta}$ and the angle from the positive, real axis around the plane is denoted by θ. When mapping poles and zeros onto the plane, poles are denoted by an "x" and zeros by an "o". The below figure shows the Z-Plane, and examples of plotting zeros and poles onto the plane can be found in the following section.
Z-Plane



Examples of Pole/Zero Plots

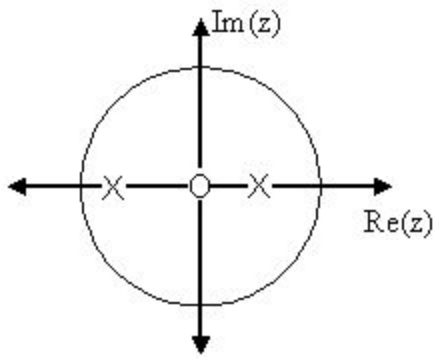
This section lists several examples of finding the poles and zeros of a transfer function and then plotting them onto the Z-Plane.

Example:
Simple Pole/Zero Plot

$$H(z) = \frac{z}{\left(z - \frac{1}{2}\right)\left(z + \frac{3}{4}\right)}$$

The zeros are: $\{0\}$
The poles are: $\left\{\frac{1}{2}, -\frac{3}{4}\right\}$

Pole/Zero Plot



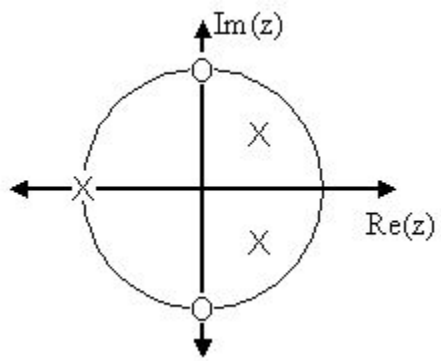
Using the zeros and poles found from the transfer function, the one zero is mapped to zero and the two poles are placed at $\frac{1}{2}$ and $-\frac{3}{4}$

Example:
Complex Pole/Zero Plot

$$H(z) = \frac{(z - i)(z + i)}{\left(z - \left(\frac{1}{2} - \frac{1}{2}i\right)\right)\left(z - \frac{1}{2} + \frac{1}{2}i\right)}$$

The zeros are: $\{i, -i\}$
The poles are: $\left\{-1, \frac{1}{2} + \frac{1}{2}i, \frac{1}{2} - \frac{1}{2}i\right\}$

Pole/Zero Plot



Using the zeros and poles found from the transfer function, the zeros are mapped to $\pm(i)$, and the poles are placed at -1, $\frac{1}{2} + \frac{1}{2}i$ and $\frac{1}{2} - \frac{1}{2}i$

Example:
Pole-Zero Cancellation

An easy mistake to make with regards to poles and zeros is to think that a function like $\frac{(s+3)(s-1)}{s-1}$ is the same as $s + 3$. In theory they are equivalent, as the pole and zero at $s = 1$ cancel each other out in what is known as **pole-zero cancellation**. However, think about what may happen if this were a transfer function of a system that was created with physical circuits. In this case, it is very unlikely that the pole and zero would remain in exactly the same place. A minor temperature change, for instance, could cause one of them to move just slightly. If this were to occur a tremendous amount of volatility is created in that area, since there is a change from infinity at the pole to zero at the zero in a very small range of signals. This

is generally a very bad way to try to eliminate a pole. A much better way is to use **control theory** to move the pole to a better place.

Note:
Repeated Poles and Zeros
It is possible to have more than one pole or zero at any given point. For instance, the discrete-time transfer function $H(z) = z^2$ will have two zeros at the origin and the continuous-time function $H(s) = \frac{1}{s^{25}}$ will have 25 poles at the origin.

MATLAB - If access to MATLAB is readily available, then you can use its functions to easily create pole/zero plots. Below is a short program that plots the poles and zeros from the above example onto the Z-Plane.

```
        % Set up vector for zeros
        z = [j ; -j];

        % Set up vector for poles
        p = [-1 ; .5+.5j ; .5-.5j];

        figure(1);
        zplane(z,p);
        title('Pole/Zero Plot for Complex
Pole/Zero Plot Example');
```

Interactive Demonstration of Poles and Zeros



Interact (when online) with a Mathematica CDF demonstrating Pole/Zero Plots. To Download, right-click and save target as .cdf.

Applications for pole-zero plots

Stability and Control theory

Now that we have found and plotted the poles and zeros, we must ask what it is that this plot gives us. Basically what we can gather from this is that the magnitude of the transfer function will be larger when it is closer to the poles and smaller when it is closer to the zeros. This provides us with a qualitative understanding of what the system does at various frequencies and is crucial to the discussion of stability.

Pole/Zero Plots and the Region of Convergence

The region of convergence (ROC) for $X(z)$ in the complex Z-plane can be determined from the pole/zero plot. Although several regions of convergence may be possible, where each one corresponds to a different impulse response, there are some choices that are more practical. A ROC can be chosen to make the transfer function causal and/or stable depending on the pole/zero plot.

Filter Properties from ROC

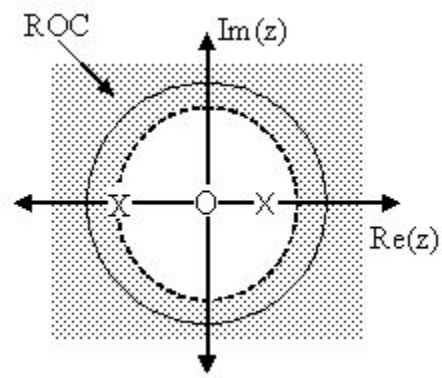
- If the ROC extends outward from the outermost pole, then the system is **causal**.
- If the ROC includes the unit circle, then the system is **stable**.

Below is a pole/zero plot with a possible ROC of the Z-transform in the Simple Pole/Zero Plot discussed earlier. The shaded region indicates the ROC chosen for the filter. From this figure, we can see that the filter will be both causal and stable since the above listed conditions are both met.

Example:

$$H(z) = \frac{z}{\left(z - \frac{1}{2}\right)\left(z + \frac{3}{4}\right)}$$

Region of Convergence for the Pole/Zero Plot



The shaded area represents the chosen ROC for the transfer function.

Frequency Response and Pole/Zero Plots

The reason it is helpful to understand and create these pole/zero plots is due to their ability to help us easily design a filter. Based on the location of the poles and zeros, the magnitude response of the filter can be quickly understood. Also, by starting with the pole/zero plot, one can design a filter and obtain its transfer function very easily.

Introduction to Random Signals and Processes

Before now, you have probably dealt strictly with the theory behind signals and systems, as well as look at some the basic characteristics of signals and systems. In doing so you have developed an important foundation; however, most electrical engineers do not get to work in this type of fantasy world. In many cases the signals of interest are very complex due to the randomness of the world around them, which leaves them noisy and often corrupted. This often causes the information contained in the signal to be hidden and distorted. For this reason, it is important to understand these random signals and how to recover the necessary information.

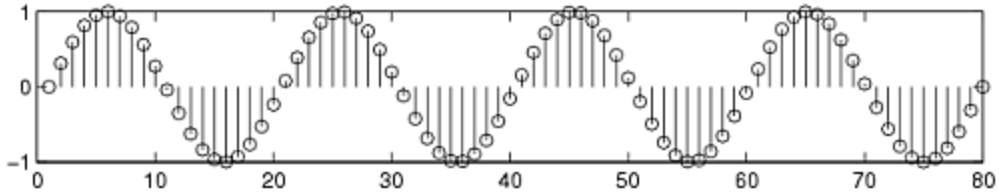
Signals: Deterministic vs. Stochastic

For this study of signals and systems, we will divide signals into two groups: those that have a fixed behavior and those that change randomly. As most of you have probably already dealt with the first type, we will focus on introducing you to random signals. Also, note that we will be dealing strictly with discrete-time signals since they are the signals we deal with in DSP and most real-world computations, but these same ideas apply to continuous-time signals.

Deterministic Signals

Most introductions to signals and systems deal strictly with **deterministic signals**. Each value of these signals are fixed and can be determined by a mathematical expression, rule, or table. Because of this, future values of any deterministic signal can be calculated from past values. For this reason, these signals are relatively easy to analyze as they do not change, and we can make accurate assumptions about their past and future behavior.

Deterministic Signal

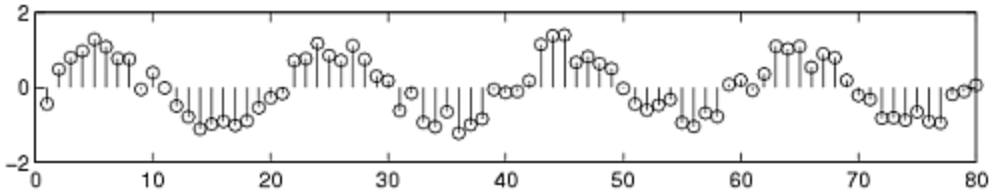


An example of a deterministic signal, the sine wave.

Stochastic Signals

Unlike deterministic signals, **stochastic signals**, or **random signals**, are not so nice. Random signals cannot be characterized by a simple, well-defined mathematical equation and their future values cannot be predicted. Rather, we must use probability and statistics to analyze their behavior. Also, because of their randomness, average values from a collection of signals are usually studied rather than analyzing one individual signal.

Random Signal



We have taken the above sine wave and added random noise to it to come up with a noisy, or random, signal. These are the types of signals that we wish to learn how to deal with so that we can recover the original sine wave.

Random Process

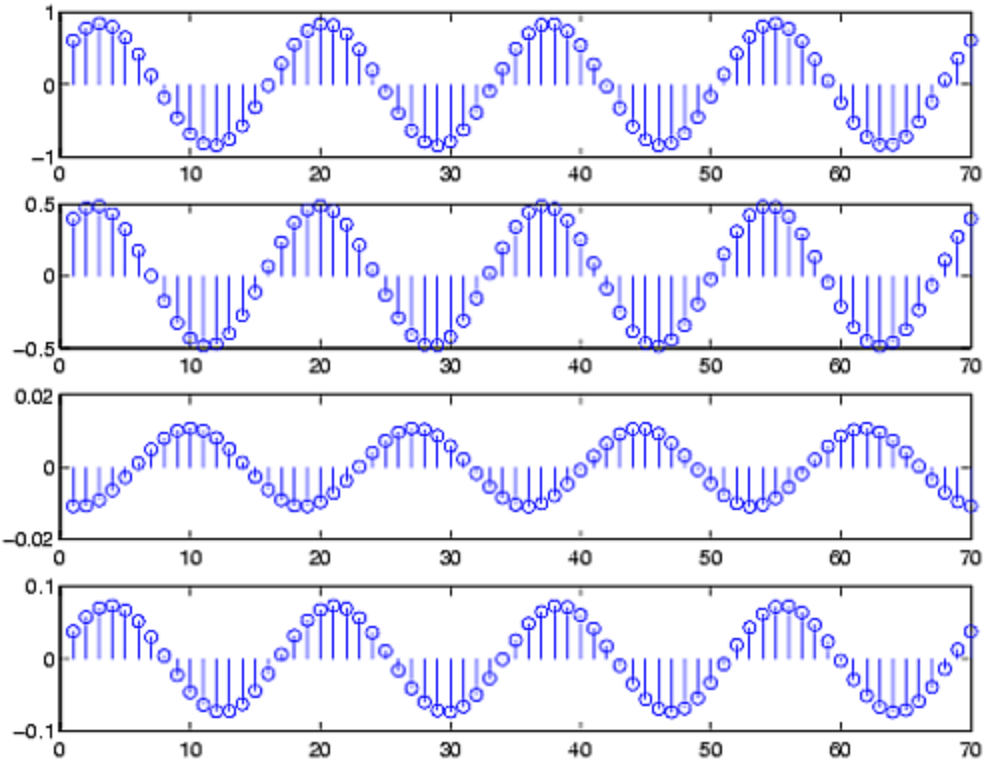
As mentioned above, in order to study random signals, we want to look at a collection of these signals rather than just one instance of that signal. This collection of signals is called a **random process**.

random process

A family or ensemble of signals that correspond to every possible outcome of a certain signal measurement. Each signal in this collection is referred to as a **realization** or **sample function** of the process.

Example:
As an example of a random process, let us look at the Random Sinusoidal Process below. We use to represent the sinusoid with a given amplitude and phase. Note that the phase and amplitude of each sinusoid is based on a random number, thus making this a random process.

Random Sinusoidal Process



A random sinusoidal process, with the amplitude and phase being random numbers.

A random process is usually denoted by or , with or used to represent an individual signal or waveform from this process.

In many notes and books, you might see the following notation and terms used to describe different types of random processes. For a **discrete random process**, sometimes just called a **random sequence**, represents time that has a finite number of values. If can take on any value of time, we have a **continuous random process**. Often times discrete and continuous refer to the amplitude of the process, and process or sequence refer to the nature of the time variable. For this study, we often just use **random process** to refer to a general collection of discrete-time signals, as seen above in [link].

Stationary and Nonstationary Random Processes

Introduction

From the definition of a random process, we know that all random processes are composed of random variables, each at its own unique point in time. Because of this, random processes have all the properties of random variables, such as mean, correlation, variances, etc.. When dealing with groups of signals or sequences it will be important for us to be able to show whether of not these statistical properties hold true for the entire random process. To do this, the concept of **stationary processes** has been developed. The general definition of a stationary process is:

stationary process
: a random process where all of its statistical properties do not vary with time

Processes whose statistical properties do change are referred to as **nonstationary**.

Understanding the basic idea of stationarity will help you to be able to follow the more concrete and mathematical definition to follow. Also, we will look at various levels of stationarity used to describe the various types of stationarity characteristics a random process can have.

Distribution and Density Functions

In order to properly define what it means to be stationary from a mathematical standpoint, one needs to be somewhat familiar with the concepts of distribution and density functions. If you can remember your statistics then feel free to skip this section!

Recall that when dealing with a single random variable, the **probability distribution function** is a simply tool used to identify the probability that our observed random variable will be less than or equal to a given number. More precisely, let X be our random variable, and let x be our given value; from this we can define the distribution function as

Equation:

$$F_x(x) = \Pr[X \leq x]$$

This same idea can be applied to instances where we have multiple random variables as well. There may be situations where we want to look at the probability of event X **and** Y both occurring. For example, below is an example of a second-order **joint distribution function**.
Equation:

$$F_x(x, y) = \Pr[X \leq x, Y \leq y]$$

While the distribution function provides us with a full view of our variable or processes probability, it is not always the most useful for calculations. Often times we will want to look at its derivative, the **probability density function (pdf)**. We define the the pdf as
Equation:

$$f_x(x) = \frac{\mathrm{d}}{\mathrm{d}\, x} F_x(x)$$

Equation:

$$f_x(x)\,\mathrm{dx} = \Pr[x < X \leq x + \mathrm{dx}]$$

[link] reveals some of the physical significance of the density function. This equations tells us the probability that our random variable falls within a given interval can be approximated by $f_x(x)$ dx. From the pdf, we can now use our knowledge of integrals to evaluate probabilities from the above approximation. Again we can also define a **joint density function** which will include multiple random variables just as was done for the distribution function. The density function is used for a variety of calculations, such as finding the expected value or proving a random variable is stationary, to name a few.

Note: The above examples explain the distribution and density functions in terms of a single random variable, X. When we are dealing with signals and random processes, remember that we will have a set of random variables where a different random variable will occur at each time instance of the random process, $X(t_k)$. In other words, the distribution and density function will also need to take into account the choice of time.

Stationarity

Below we will now look at a more in depth and mathematical definition of a stationary process. As was mentioned previously, various levels of stationarity exist and we will look at the most common types.

First-Order Stationary Process

A random process is classified as **first-order stationary** if its first-order probability density function remains equal regardless of any shift in time to its time origin. If we let x_{t_1} represent a given value at time t_1, then we define a first-order stationary as one that satisfies the following equation:
Equation:

$$f_x(x_{t_1}) = f_x(x_{t_1+\tau})$$

The physical significance of this equation is that our density function, $f_x(x_{t_1})$, is completely independent of t_1 and thus any time shift, τ.

The most important result of this statement, and the identifying characteristic of any first-order stationary process, is the fact that the mean is a constant, independent of any time shift. Below we show the results for a random process, X, that is a discrete-time signal, $x[n]$.
Equation:

$$\begin{aligned} X &= m_x[n] \\ &= E[x[n]] \\ &= \text{constant (independent of n)} \end{aligned}$$

Second-Order and Strict-Sense Stationary Process

A random process is classified as **second-order stationary** if its second-order probability density function does not vary over any time shift applied to both values. In other words, for values x_{t_1} and x_{t_2} then we will have the following be equal for an arbitrary time shift τ.
Equation:

$$f_x(x_{t_1}, x_{t_2}) = f_x(x_{t_1+\tau}, x_{t_2+\tau})$$

From this equation we see that the absolute time does not affect our functions, rather it only really depends on the time difference between the two variables. Looked at another way, this equation can be described as
Equation:

$$\Pr[X(t_1) \leq x_1, X(t_2) \leq x_2] = \Pr[X(t_1 + \tau) \leq x_1, X(t_2 + \tau) \leq x_2]$$

These random processes are often referred to as **strict sense stationary (SSS)** when **all** of the distribution functions of the process are unchanged regardless of the time shift applied to them.

For a second-order stationary process, we need to look at the autocorrelation function to see its most important property. Since we have already stated that a second-order stationary process depends only on the time difference, then all of these types of processes have the following property:
Equation:

$$\begin{aligned} R_{xx}(t, t+\tau) &= E[X(t+\tau)] \\ &= R_{xx}(\tau) \end{aligned}$$

Wide-Sense Stationary Process

As you begin to work with random processes, it will become evident that the strict requirements of a SSS process is more than is often necessary in order to adequately approximate our calculations on random processes. We define a final type of stationarity, referred to as **wide-sense stationary (WSS)**, to have slightly more relaxed requirements but ones that are still enough to provide us with adequate results. In order to be WSS a random process only needs to meet the following two requirements.

1. $X = E[x[n]] = \text{constant}$
2. $E[X(t+\tau)] = R_{xx}(\tau)$

Note that a second-order (or SSS) stationary process will always be WSS; however, the reverse will not always hold true.

Random Processes: Mean and Variance

In order to study the characteristics of a random process, let us look at some of the basic properties and operations of a random process. Below we will focus on the operations of the random signals that compose our random processes. We will denote our random process with X and a random variable from a random process or signal by x.

Mean Value

Finding the average value of a set of random signals or random variables is probably the most fundamental concepts we use in evaluating random processes through any sort of statistical method. **The mean of a random process is the average of all realizations of that process.** In order to find this average, we must look at a random signal over a range of time (possible values) and determine our average from this set of values. The **mean**, or average, of a random process, $x(t)$, is given by the following equation:
Equation:

$$\begin{aligned} m_x(t) &= \mu_x(t) \\ &= \overline{X} \\ &= E[X] \\ &= \int_{-\infty}^{\infty} x f(x)\, \mathrm{d}\, x \end{aligned}$$

This equation may seem quite cluttered at first glance, but we want to introduce you to the various notations used to represent the mean of a random signal or process. Throughout texts and other readings, remember that these will all equal the same thing. The symbol, $\mu_x(t)$, and the X with a bar over it are often used as a short-hand to represent an average, so you might see it in certain textbooks. The other important notation used is, $E[X]$, which represents the "expected value of X" or the mathematical expectation. This notation is very common and will appear again.

If the random variables, which make up our random process, are discrete or quantized values, such as in a binary process, then the integrals become

summations over all the possible values of the random variable. In this case, our expected value becomes
Equation:

$$E[x[n]] = \sum_{x} \alpha \Pr[x[n] = \alpha]$$

If we have two random signals or variables, their averages can reveal how the two signals interact. If the product of the two individual averages of both signals do **not** equal the average of the product of the two signals, then the two signals are said to be **linearly independent**, also referred to as **uncorrelated**.

In the case where we have a random process in which only one sample can be viewed at a time, then we will often not have all the information available to calculate the mean using the density function as shown above. In this case we must estimate the mean through the time-average mean, discussed later. For fields such as signal processing that deal mainly with discrete signals and values, then these are the averages most commonly used.

Properties of the Mean

- The expected value of a constant, α, is the constant:
  Equation:

$$E[\alpha] = \alpha$$

- Adding a constant, α, to each term increases the expected value by that constant:
  Equation:

$$E[X + \alpha] = E[X] + \alpha$$

- Multiplying the random variable by a constant, α, multiplies the expected value by that constant.

Equation:

$$E[\alpha X] = \alpha E[X]$$

- The expected value of the sum of two or more random variables, is the sum of each individual expected value.
  Equation:

$$E[X + Y] = E[X] + E[Y]$$

Mean-Square Value

If we look at the second **moment** of the term (we now look at x^2 in the integral), then we will have the **mean-square value** of our random process. As you would expect, this is written as
Equation:

$$\begin{aligned} \overline{X^2} &= E\left[X^2\right] \\ &= \int_{-\infty}^{\infty} x^2 f(x)\ \mathrm{d}\ x \end{aligned}$$

This equation is also often referred to as the **average power** of a process or signal.

Variance

Now that we have an idea about the average value or values that a random process takes, we are often interested in seeing just how spread out the different random values might be. To do this, we look at the **variance** which is a measure of this spread. The variance, often denoted by σ^2, is written as follows:
Equation:

$$\begin{aligned}\sigma^2 &= \operatorname{Var}(X) \\ &= E\left[(X - E[X])^2\right] \\ &= \int_{-\infty}^{\infty} \left(x - \overline{X}\right)^2 f(x)\, \mathrm{d}\, x\end{aligned}$$

Using the rules for the expected value, we can rewrite this formula as the following form, which is commonly seen:
Equation:

$$\begin{aligned}\sigma^2 &= \overline{X^2} - \left(\overline{X}\right)^2 \\ &= E\left[X^2\right] - (E[X])^2\end{aligned}$$

Standard Deviation

Another common statistical tool is the standard deviation. Once you know how to calculate the variance, the standard deviation is simply **the square root of the variance**, or σ.

Properties of Variance

- The variance of a constant, α, equals zero:
  Equation:

$$\begin{aligned}\operatorname{Var}(\alpha) &= \sigma(\alpha)^2 \\ &= 0\end{aligned}$$

- Adding a constant, α, to a random variable does not affect the variance because the mean increases by the same value:
  Equation:

$$\begin{aligned}\operatorname{Var}(X+\alpha) &= \sigma(X+\alpha)^2 \\ &= \sigma(X)^2\end{aligned}$$

- Multiplying the random variable by a constant, α, increases the variance by the square of the constant:
  Equation:

$$\begin{aligned}\operatorname{Var}(\alpha X) &= \sigma(\alpha X)^2 \\ &= \alpha^2\sigma(X)^2\end{aligned}$$

- The variance of the sum of two random variables only equals the sum of the variances if the variable are **independent**.
  Equation:

$$\begin{aligned}\operatorname{Var}(X+Y) &= \sigma(X+Y)^2 \\ &= \sigma(X)^2+\sigma(Y)^2\end{aligned}$$

  Otherwise, if the random variable are **not** independent, then we must also include the covariance of the product of the variables as follows:
  Equation:

$$\operatorname{Var}(X+Y) = \sigma(X)^2 + 2\operatorname{Cov}(X,Y) + \sigma(Y)^2$$

Time Averages

In the case where we can not view the entire ensemble of the random process, we must use time averages to estimate the values of the mean and variance for the process. Generally, this will only give us acceptable results for independent and **ergodic** processes, meaning those processes in which each signal or member of the process seems to have the same statistical behavior as the entire process. The time averages will also only be taken over a finite interval since we will only be able to see a finite part of the sample.

Estimating the Mean

For the ergodic random process, $x(t)$, we will estimate the mean using the time averaging function defined as
Equation:

$$\begin{aligned} \overline{X} &= E[X] \\ &= \tfrac{1}{T}\int_0^T X(t)\ \mathrm{d}\ t \end{aligned}$$

However, for most real-world situations we will be dealing with discrete values in our computations and signals. We will represent this mean as
Equation:

$$\begin{aligned} \overline{X} &= E[X] \\ &= \tfrac{1}{N}\sum_{n=1}^{N} X[n] \end{aligned}$$

Estimating the Variance

Once the mean of our random process has been estimated then we can simply use those values in the following variance equation (introduced in one of the above sections)
Equation:

$${\sigma_x}^2 = \overline{X^2} - \left(\overline{X}\right)^2$$

Example

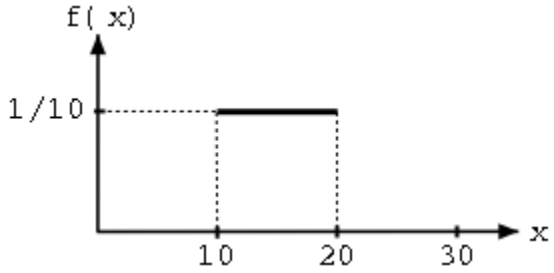
Let us now look at how some of the formulas and concepts above apply to a simple example. We will just look at a single, continuous random variable for this example, but the calculations and methods are the same for a

random process. For this example, we will consider a random variable having the probability density function described below and shown in [link].

Equation:

$$f(x) = \begin{cases} \frac{1}{10} & \text{if } 10 \leq x \leq 20 \\ 0 & \text{otherwise} \end{cases}$$

Probability Density Function



A uniform probability density function.

First, we will use [link] to solve for the mean value.

Equation:

$$\begin{aligned} \overline{X} &= \int_{10}^{20} x \frac{1}{10} \, \mathrm{d}\, x \\ &= \frac{1}{10} \frac{x^2}{2} \Big|_{x=10}^{20} \\ &= \frac{1}{10} (200 - 50) \\ &= 15 \end{aligned}$$

Using [link] we can obtain the mean-square value for the above density function.

Equation:

$$\begin{aligned}
\overline{X^2} &= \int_{10}^{20} x^2 \tfrac{1}{10} \, \mathrm{d}\, x \\
&= \tfrac{1}{10} \tfrac{x^3}{3} \Big|_{x=10}^{20} \\
&= \tfrac{1}{10} \left(\tfrac{8000}{3} - \tfrac{1000}{3} \right) \\
&= 233.33
\end{aligned}$$

And finally, let us solve for the variance of this function.
Equation:

$$\begin{aligned}
\sigma^2 &= \overline{X^2} - \left(\overline{X} \right)^2 \\
&= 233.33 - 15^2 \\
&= 8.33
\end{aligned}$$

Correlation and Covariance of a Random Signal

When we take the expected value, or average, of a random process, we measure several important characteristics about how the process behaves in general. This proves to be a very important observation. However, suppose we have several random processes measuring different aspects of a system. The relationship between these different processes will also be an important observation. The covariance and correlation are two important tools in finding these relationships. Below we will go into more details as to what these words mean and how these tools are helpful. Note that much of the following discussions refer to just random variables, but keep in mind that these variables can represent random signals or random processes.

Covariance

To begin with, when dealing with more than one random process, it should be obvious that it would be nice to be able to have a number that could quickly give us an idea of how similar the processes are. To do this, we use the **covariance**, which is analogous to the variance of a single variable.

Covariance
: A measure of how much the deviations of two or more variables or processes match.

For two processes, X and Y, if they are **not** closely related then the covariance will be small, and if they are similar then the covariance will be large. Let us clarify this statement by describing what we mean by "related" and "similar." Two processes are "closely related" if their distribution spreads are almost equal and they are around the same, or a very slightly different, mean.

Mathematically, covariance is often written as σ_{xy} and is defined as
Equation:

$$\begin{aligned} \operatorname{cov}(X,Y) &= \sigma_{xy} \\ &= E\Big[\Big(X-X\Big)\Big(Y-Y\Big)\Big] \end{aligned}$$

This can also be reduced and rewritten in the following two forms:
Equation:

$$\sigma_{xy} = (xy) - \bar{x}\bar{y}$$

Equation:

$$\sigma_{xy} = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \left(X - X\right) \left(Y - Y\right) f(x, y) \, \mathrm{d}\, x \, \mathrm{d}\, y$$

Useful Properties

- If X and Y are independent and uncorrelated or one of them has zero mean value, then

$$\sigma_{xy} = 0$$

- If X and Y are orthogonal, then

$$\sigma_{xy} = -\left(E[X]E[Y]\right)$$

- The covariance is symmetric

$$\operatorname{cov}\left(X, Y\right) = \operatorname{cov}\left(Y, X\right)$$

- Basic covariance identity

$$\operatorname{cov}\left(X + Y, Z\right) = \operatorname{cov}\left(X, Z\right) + \operatorname{cov}\left(Y, Z\right)$$

- Covariance of equal variables

$$\operatorname{cov}\left(X, X\right) = \operatorname{Var}\left(X\right)$$

Correlation

For anyone who has any kind of statistical background, you should be able to see that the idea of dependence/independence among variables and signals plays an important role when dealing with random processes. Because of this, the **correlation** of two variables provides us with a measure of how the two variables affect one another.

Correlation

A measure of how much one random variable depends upon the other.

This measure of association between the variables will provide us with a clue as to how well the value of one variable can be predicted from the value of the other. The correlation is equal to the average of the product of two random variables and is defined as

Equation:

$$\begin{aligned} \operatorname{cor}(X,Y) &= E[XY] \\ &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} xyf(x,y)\,\mathrm{d}\,x\,\mathrm{d}\,y \end{aligned}$$

Correlation Coefficient

It is often useful to express the correlation of random variables with a range of numbers, like a percentage. For a given set of variables, we use the **correlation coefficient** to give us the linear relationship between our variables. The correlation coefficient of two variables is defined in terms of their covariance and standard deviations, denoted by σ_x, as seen below

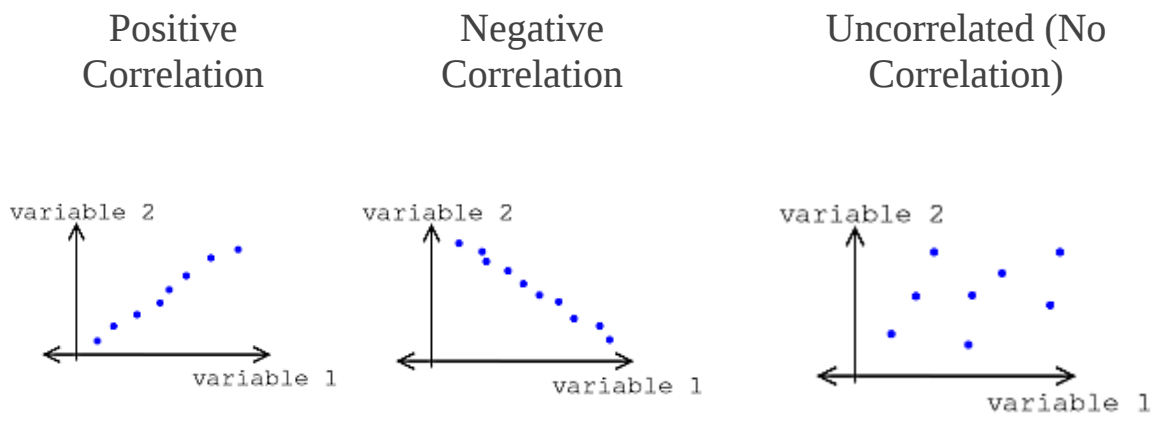
Equation:

$$\rho = \frac{\operatorname{cov}(X,Y)}{\sigma_x \sigma_y}$$

where we will always have

$$-1 \leq \rho \leq 1$$

This provides us with a quick and easy way to view the correlation between our variables. If there is no relationship between the variables then the correlation coefficient will be zero and if there is a perfect positive match it will be one. If there is a perfect inverse relationship, where one set of variables increases while the other decreases, then the correlation coefficient will be negative one. This type of correlation is often referred to more specifically as the **Pearson's Correlation Coefficient**,or Pearson's Product Moment Correlation.



Types of Correlation

Note: So far we have dealt with correlation simply as a number relating the relationship between any two variables. However, since our goal will be to relate random processes to each other, which deals with signals as a function of time, we will want to continue this study by looking at **correlation functions**.

Example

Now let us take just a second to look at a simple example that involves calculating the covariance and correlation of two sets of random numbers. We are given the following data sets:

$$x = \{3, 1, 6, 3, 4\}$$

$$y = \{1, 5, 3, 4, 3\}$$

To begin with, for the covariance we will need to find the expected value, or mean, of x and y.

$$\bar{x} = \frac{1}{5}\left(3 + 1 + 6 + 3 + 4\right) = 3.4$$

$$\bar{y} = \frac{1}{5}\left(1 + 5 + 3 + 4 + 3\right) = 3.2$$

$$xy = \frac{1}{5}\left(3 + 5 + 18 + 12 + 12\right) = 10$$

Next we will solve for the standard deviations of our two sets using the formula below (for a review click here).

$$\sigma = \sqrt{E\left[\left(X - E[X]\right)^2\right]}$$

$$\sigma_x = \sqrt{\frac{1}{5}\left(0.16 + 5.76 + 6.76 + 0.16 + 0.36\right)} = 1.625$$

$$\sigma_y = \sqrt{\frac{1}{6}\left(4.84 + 3.24 + 0.04 + 0.64 + 0.04\right)} = 1.327$$

Now we can finally calculate the covariance using one of the two formulas found above. Since we calculated the three means, we will use that formula since it will be much simpler.

$$\sigma_{xy} = 10 - 3.4 \times 3.2 = -0.88$$

And for our last calculation, we will solve for the correlation coefficient, ρ.

$$\rho = \frac{-0.88}{1.625 \times 1.327} = -0.408$$

Matlab Code for Example

The above example can be easily calculated using Matlab. Below I have included the code to find all of the values above.

```
x = [3 1 6 3 4];
y = [1 5 3 4 3];

mx = mean(x)
my = mean(y)
mxy = mean(x.*y)

% Standard Dev. from built-in Matlab
Functions
std(x,1)
std(y,1)

% Standard Dev. from Equation Above
(same result as std(?,1))
sqrt( 1/5 * sum((x-mx).^2))
sqrt( 1/5 * sum((y-my).^2))

cov(x,y,1)

corrcoef(x,y)
```

Autocorrelation of Random Processes

Before diving into a more complex statistical analysis of random signals and processes, let us quickly review the idea of correlation. Recall that the correlation of two signals or variables is the expected value of the product of those two variables. Since our focus will be to discover more about a random process, a collection of random signals, then imagine us dealing with two samples of a random process, where each sample is taken at a different point in time. Also recall that the key property of these random processes is that they are now functions of time; imagine them as a collection of signals. The expected value of the product of these two variables (or samples) will now depend on how quickly they change in regards to **time**. For example, if the two variables are taken from almost the same time period, then we should expect them to have a high correlation. We will now look at a correlation function that relates a pair of random variables from the same process to the time separations between them, where the argument to this correlation function will be the time difference. For the correlation of signals from two different random process, look at the crosscorrelation function.

Autocorrelation Function

The first of these correlation functions we will discuss is the **autocorrelation**, where each of the random variables we will deal with come from the same random process.

Autocorrelation
: the expected value of the product of a random variable or signal realization with a time-shifted version of itself

With a simple calculation and analysis of the autocorrelation function, we can discover a few important characteristics about our random process. These include:

1. How quickly our random signal or processes changes with respect to the time function

2. Whether our process has a periodic component and what the expected frequency might be

As was mentioned above, the autocorrelation function is simply the expected value of a product. Assume we have a pair of random variables from the same process, $X_1 = X(t_1)$ and $X_2 = X(t_2)$, then the autocorrelation is often written as
Equation:

$$\begin{aligned} R_{\text{xx}}(t_1, t_2) &= E[X_1 X_2] \\ &= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} x_1 x_2 f(x_1, x_2) \, \mathrm{d}\, x_2 \, \mathrm{d}\, x_1 \end{aligned}$$

The above equation is valid for stationary and nonstationary random processes. For stationary processes, we can generalize this expression a little further. Given a wide-sense stationary processes, it can be proven that the expected values from our random process will be independent of the origin of our time function. Therefore, we can say that our autocorrelation function will depend on the time difference and not some absolute time. For this discussion, we will let $\tau = t_2 - t_1$, and thus we generalize our autocorrelation expression as
Equation:

$$\begin{aligned} R_{\text{xx}}(t, t+\tau) &= R_{\text{xx}}(\tau) \\ &= E[X(t) X(t+\tau)] \end{aligned}$$

for the continuous-time case. In most DSP course we will be more interested in dealing with real signal sequences, and thus we will want to look at the discrete-time case of the autocorrelation function. The formula below will prove to be more common and useful than [link]:
Equation:

$$R_{\text{xx}}[n, n+m] = \sum_{n=-\infty}^{\infty} x[n] x[n+m]$$

And again we can generalize the notation for our autocorrelation function as **Equation:**

$$\begin{aligned} R_{\text{xx}}[n, n+m] &= R_{\text{xx}}[m] \\ &= E[X[n]X[n+m]] \end{aligned}$$

Properties of Autocorrelation

Below we will look at several properties of the autocorrelation function that hold for **stationary** random processes.

- Autocorrelation is an even function for τ

$$R_{\text{xx}}(\tau) = R_{\text{xx}}(-\tau)$$

- The mean-square value can be found by evaluating the autocorrelation where $\tau = 0$, which gives us

$$R_{\text{xx}}(0) = X^2$$

- The autocorrelation function will have its largest value when $\tau = 0$. This value can appear again, for example in a periodic function at the values of the equivalent periodic points, but will never be exceeded.

$$R_{\text{xx}}(0) \geq |R_{\text{xx}}(\tau)|$$

- If we take the autocorrelation of a period function, then $R_{\text{xx}}(\tau)$ will also be periodic with the same frequency.

Estimating the Autocorrleation with Time-Averaging

Sometimes the whole random process is not available to us. In these cases, we would still like to be able to find out some of the characteristics of the

stationary random process, even if we just have part of one sample function. In order to do this we can **estimate** the autocorrelation from a given interval, 0 to T seconds, of the sample function.
Equation:

$$\check{R}_{\mathrm{xx}}(\tau) = \frac{1}{T-\tau} \int_0^{T-\tau} x(t)x(t+\tau)\ \mathrm{d}\,t$$

However, a lot of times we will not have sufficient information to build a complete continuous-time function of one of our random signals for the above analysis. If this is the case, we can treat the information we do know about the function as a discrete signal and use the discrete-time formula for estimating the autocorrelation.
Equation:

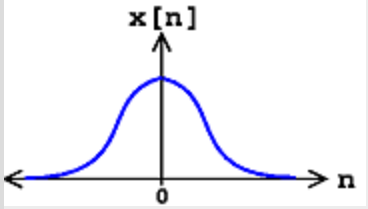
$$\check{R}_{\mathrm{xx}}[m] = \frac{1}{N-m} \sum_{n=0}^{N-m-1} x[n]x[n+m]$$

Examples

Below we will look at a variety of examples that use the autocorrelation function. We will begin with a simple example dealing with Gaussian White Noise (GWN) and a few basic statistical properties that will prove very useful in these and future calculations.

Example:
We will let $x[n]$ represent our GWN. For this problem, it is important to remember the following fact about the mean of a GWN function:

$$E[x[n]] = 0$$



Gaussian density function. By examination, can easily see that the above statement is true - the mean equals zero.

Along with being **zero-mean**, recall that GWN is always **independent**. With these two facts, we are now ready to do the short calculations required to find the autocorrelation.

$$R_{\text{xx}}[n, n+m] = E[x[n]x[n+m]]$$

Since the function, $x[n]$, is independent, then we can take the product of the individual expected values of both functions.

$$R_{\text{xx}}[n, n+m] = E[x[n]]E[x[n+m]]$$

Now, looking at the above equation we see that we can break it up further into two conditions: one when m and n are equal and one when they are not equal. When they are equal we can combine the expected values. We are left with the following piecewise function to solve:

$$R_{\text{xx}}[n, n+m] = \begin{cases} E[x[n]]E[x[n+m]] & \text{if } \ m \neq 0 \\ E\left[x^2[n]\right] & \text{if } \ m = 0 \end{cases}$$

We can now solve the two parts of the above equation. The first equation is easy to solve as we have already stated that the expected value of $x[n]$ will be zero. For the second part, you should recall from statistics that the

expected value of the square of a function is equal to the variance. Thus we get the following results for the autocorrelation:

$$R_{\text{xx}}[n, n+m] = \begin{cases} 0 \ \text{ if } \ m \neq 0 \\ \sigma^2 \ \text{ if } \ m = 0 \end{cases}$$

Or in a more concise way, we can represent the results as

$$R_{\text{xx}}[n, n+m] = \sigma^2 \delta[m]$$

Crosscorrelation of Random Processes

Before diving into a more complex statistical analysis of random signals and processes, let us quickly review the idea of correlation. Recall that the correlation of two signals or variables is the expected value of the product of those two variables. Since our main focus is to discover more about random processes, a collection of random signals, we will deal with two random processes in this discussion, where in this case we will deal with samples from two **different** random processes. We will analyze the expected value of the product of these two variables and how they correlate to one another, where the argument to this correlation function will be the time difference. For the correlation of signals from the same random process, look at the autocorrelation function.

Crosscorrelation Function

When dealing with multiple random processes, it is also important to be able to describe the relationship, if any, between the processes. For example, this may occur if more than one random signal is applied to a system. In order to do this, we use the **crosscorrelation function**, where the variables are instances from two different wide sense stationary random processes.

Crosscorrelation
: if two processes are wide sense stationary, the expected value of the product of a random variable from one random process with a time-shifted, random variable from a different random process

Looking at the generalized formula for the crosscorrelation, we will represent our two random processes by allowing $U = U(t)$ and $V = V(t - \tau)$. We will define the crosscorrelation function as
Equation:

$$\begin{aligned} R_{uv}(t, t-\tau) &= E[UV] \\ &= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} uvf(u,v) \, \mathrm{d}\, v \, \mathrm{d}\, u \end{aligned}$$

Just as the case with the autocorrelation function, if our input and output, denoted as $U(t)$ and $V(t)$, are at least jointly wide sense stationary, then the crosscorrelation does not depend on absolute time; it is just a function of the time difference. This means we can simplify our writing of the above function as

Equation:

$$R_{uv}(\tau) = E[UV]$$

or if we deal with two real signal sequences, $x[n]$ and $y[n]$, then we arrive at a more commonly seen formula for the discrete crosscorrelation function. See the formula below and notice the similarities between it and the convolution of two signals:

Equation:

$$\begin{aligned} R_{xy}(n, n-m) &= R_{xy}(m) \\ &= \sum_{n=-\infty}^{\infty} x[n]y[n-m] \end{aligned}$$

Properties of Crosscorrelation

Below we will look at several properties of the crosscorrelation function that hold for two **wide sense stationary (WSS)** random processes.

- Crosscorrelation is **not** an even function; however, it does have a unique symmetry property:
  Equation:

  $$R_{xy}(-\tau) = R_{yx}(\tau)$$

- The maximum value of the crosscorrelation is not always when the shift equals zero; however, we can prove the following property revealing to us what value the maximum cannot exceed.
  Equation:

$$|R_{xy}(\tau)| \leq \sqrt{R_{xx}(0)R_{yy}(0)}$$

- When two random processes are statistically independent then we have **Equation:**

$$R_{xy}(\tau) = R_{yx}(\tau)$$

Examples

Exercise:

Problem:

Let us begin by looking at a simple example showing the relationship between two sequences. Using [link], find the crosscorrelation of the sequences

$$x[n] = \{\ldots, 0, 0, 2, -3, 6, 1, 3, 0, 0, \ldots\}$$

$$y[n] = \{\ldots, 0, 0, 1, -2, 4, 1, -3, 0, 0, \ldots\}$$

for each of the following possible time shifts: $m = \{0, 3, -1\}$.

Solution:

1. For $m = 0$, we should begin by finding the product sequence $s[n] = x[n]y[n]$. Doing this we get the following sequence:

   $$s[n] = \{\ldots, 0, 0, 2, 6, 24, 1, -9, 0, 0, \ldots\}$$

   and so from the sum in our crosscorrelation function we arrive at the answer of

   $$R_{xy}(0) = 22$$

2. For $m = 3$, we will approach it the same was we did above; however, we will now shift $y[n]$ to the right. Then we can find the

product sequence $s[n] = x[n]y[n-3]$, which yields

$$s[n] = \{\ldots, 0, 0, 0, 0, 0, 1, -6, 0, 0, \ldots\}$$

and from the crosscorrelation function we arrive at the answer of

$$R_{xy}(3) = -6$$

3. For $m = -1$, we will again take the same approach; however, we will now shift $y[n]$ to the left. Then we can find the product sequence $s[n] = x[n]y[n+1]$, which yields

$$s[n] = \{\ldots, 0, 0, -4, -12, 6, -3, 0, 0, 0, \ldots\}$$

and from the crosscorrelation function we arrive at the answer of

$$R_{xy}(-1) = -13$$

DFT Definition and Properties

DFT

The discrete Fourier transform (DFT) is the primary transform used for numerical computation in digital signal processing. It is very widely used for spectrum analysis, fast convolution, and many other applications. The DFT transforms N discrete-time samples to the same number of discrete frequency samples, and is defined as
Equation:

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-\left(i \frac{2\pi nk}{N}\right)}$$

The DFT is widely used in part because it can be computed very efficiently using fast Fourier transform (FFT) algorithms.

IDFT

The inverse DFT (IDFT) transforms N discrete-frequency samples to the same number of discrete-time samples. The IDFT has a form very similar to the DFT,
Equation:

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{i \frac{2\pi nk}{N}}$$

and can thus also be computed efficiently using FFTs.

DFT and IDFT properties

Periodicity

Due to the N-sample periodicity of the complex exponential basis functions $e^{i \frac{2\pi nk}{N}}$ in the DFT and IDFT, the resulting transforms are also periodic with N samples.

$$X(k + N) = X(k)$$

$$x(n) = x(n + N)$$

Circular Shift

A shift in time corresponds to a phase shift that is linear in frequency. Because of the periodicity induced by the DFT and IDFT, the shift is **circular**, or modulo N samples.

$$x((n-m) \bmod N) \quad X(k)e^{-\left(i\frac{2\pi km}{N}\right)}$$

The modulus operator $p \bmod N$ means the **remainder** of p when divided by N. For example,

$$9 \bmod 5 = 4$$

and

$$-1 \bmod 5 = 4$$

Time Reversal

$$x((-n) \bmod N) = x((N-n) \bmod N) \quad X((N-k) \bmod N) = X((-k) \bmod N)$$

Note: time-reversal maps $0 \quad 0$, $1 \quad N-1$, $2 \quad N-2$, etc. as illustrated in the figure below.

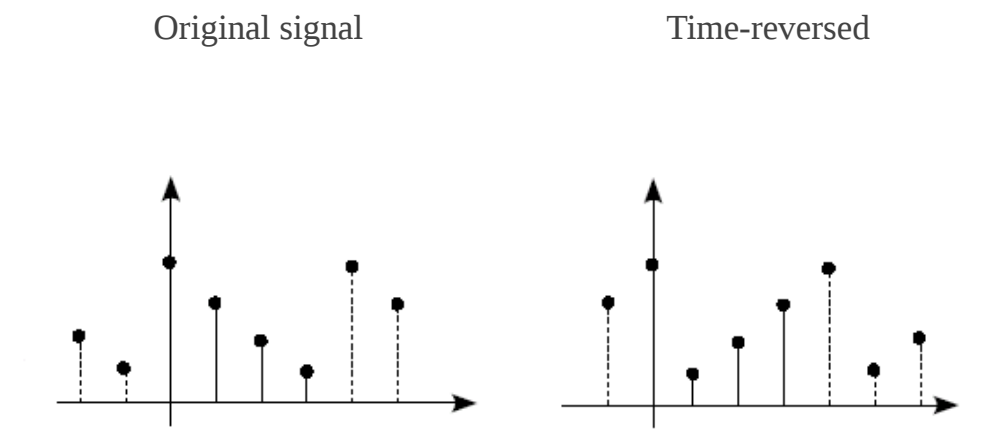


Illustration of circular time-reversal

Complex Conjugate

$$x(n) \quad X((-k) \bmod N)$$

Circular Convolution Property

Circular convolution is defined as

$$x(n)*h(n) \doteq \sum_{m=0}^{N-1} x(m)x((n-m) \bmod N)$$

Circular convolution of two discrete-time signals corresponds to multiplication of their DFTs:

$$x(n)*h(n) \quad X(k)H(k)$$

Multiplication Property

A similar property relates multiplication in time to circular convolution in frequency.

$$x(n)h(n) \quad \frac{1}{N}X(k)*H(k)$$

Parseval's Theorem

Parseval's theorem relates the energy of a length-N discrete-time signal (or one period) to the energy of its DFT.

$$\sum_{n=0}^{N-1} \left(|x(n)|\right)^2 = \frac{1}{N} \sum_{k=0}^{N-1} \left(|X(k)|\right)^2$$

Symmetry

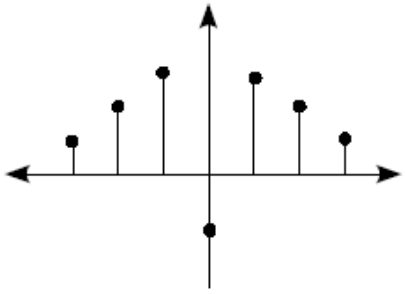
The continuous-time Fourier transform, the DTFT, and DFT are all defined as transforms of complex-valued data to complex-valued spectra. However, in practice signals are often real-valued. The DFT of a real-valued discrete-time signal has a

special symmetry, in which the real part of the transform values are **DFT even symmetric** and the imaginary part is **DFT odd symmetric**, as illustrated in the equation and figure below.

$x(n)$ real $\quad X(k) = X((N-k) \bmod N)$ (This implies $X(0)$, $X\left(\frac{N}{2}\right)$ are real-valued.)

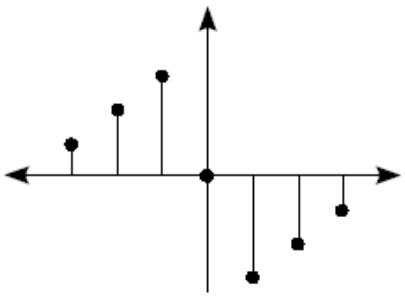
DFT symmetry of real-valued signal

Real part of X(k) is even



Even-symmetry in DFT sense

Imaginary part of X(k) is odd



Odd-symmetry in DFT sense

Spectrum Analysis Using the Discrete Fourier Transform

Discrete-Time Fourier Transform

The Discrete-Time Fourier Transform (DTFT) is the primary theoretical tool for understanding the frequency content of a discrete-time (sampled) signal. The DTFT is defined as
Equation:

$$X(\omega) = \sum_{n=-\infty}^{\infty} x(n) e^{-(i\omega n)}$$

The inverse DTFT (IDTFT) is defined by an integral formula, because it operates on a continuous-frequency DTFT spectrum:
Equation:

$$x(n) = \frac{1}{2\pi} \int_{-\pi}^{\pi} X(k) e^{i\omega n} \, \mathrm{d}\, \omega$$

The DTFT is very useful for theory and analysis, but is not practical for numerically computing a spectrum digitally, because

1. infinite time samples means
    - infinite computation
    - infinite delay
2. The transform is continuous in the discrete-time frequency, ω

For practical computation of the frequency content of real-world signals, the Discrete Fourier Transform (DFT) is used.

Discrete Fourier Transform

The DFT transforms N samples of a discrete-time signal to the same number of discrete frequency samples, and is defined as
Equation:

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-\frac{i 2\pi n k}{N}}$$

The DFT is invertible by the inverse discrete Fourier transform (IDFT):
Equation:

$$x(n) = \frac{1}{\mathrm{N}} \sum_{k=0}^{N-1} X(k) e^{i \frac{2\pi n k}{N}}$$

The DFT and IDFT are a self-contained, one-to-one transform pair for a length-N discrete-time signal. (That is, the DFT is not **merely** an approximation to the DTFT as discussed next.) However, the DFT is very often used as a practical approximation to the DTFT.

Relationships Between DFT and DTFT

DFT and Discrete Fourier Series

The DFT gives the discrete-time Fourier series coefficients of a periodic sequence ($x(n) = x(n+N)$) of period N samples, or
Equation:

$$X(\omega) = \frac{2\pi}{N} \sum X(k) \delta\left(\omega - \frac{2\pi k}{N}\right)$$

as can easily be confirmed by computing the inverse DTFT of the corresponding line spectrum:
Equation:

$$\begin{aligned} x(n) &= \tfrac{1}{2\pi} \int_{-\pi}^{\pi} \left(\tfrac{2\pi}{N} \sum X(k) \delta\left(\omega - \tfrac{2\pi k}{N}\right)\right) e^{i\omega n} \, \mathrm{d}\, \omega \\ &= \tfrac{1}{\mathrm{N}} \textstyle\sum_{k=0}^{N-1} X(k) e^{i \frac{2\pi n k}{N}} \\ &= \mathrm{IDFT}\left(X(k)\right) \\ &= x(n) \end{aligned}$$

The DFT can thus be used to **exactly** compute the relative values of the N line spectral components of the DTFT of any periodic discrete-time sequence with an integer-length period.

DFT and DTFT of finite-length data

When a discrete-time sequence happens to equal zero for all samples except for those between 0 and $N-1$, the infinite sum in the DTFT equation becomes the same as the finite sum from 0 to $N-1$ in the DFT equation. By matching the arguments in the exponential terms, we observe that the DFT values **exactly** equal the DTFT for specific DTFT frequencies $\omega_k = \frac{2\pi k}{N}$. That is, the DFT computes exact samples of the DTFT at N equally spaced frequencies $\omega_k = \frac{2\pi k}{N}$, or

$$X\left(\omega_k = \frac{2\pi k}{N}\right) = \sum_{n=-\infty}^{\infty} x(n) e^{-(i\omega_k n)} = \sum_{n=0}^{N-1} x(n) e^{-\frac{i2\pi nk}{N}} = X(k)$$

DFT as a DTFT approximation

In most cases, the signal is neither exactly periodic nor truly of finite length; in such cases, the DFT of a finite block of N consecutive discrete-time samples does **not** exactly equal samples of the DTFT at specific frequencies. Instead, the DFT gives frequency samples of a windowed (truncated) DTFT

$$\hat{X}\left(\omega_k = \frac{2\pi k}{N}\right) = \sum_{n=0}^{N-1} x(n) e^{-(i\omega_k n)} = \sum_{n=-\infty}^{\infty} x(n) w(n) e^{-(i\omega_k n)} = X(k)$$

where $w(n) = \begin{cases} 1 & \text{if } 0 \le n < N \\ 0 & \text{if else} \end{cases}$ Once again, $X(k)$ **exactly** equals $X(\omega_k)$ a DTFT frequency sample only when $\forall n, n \notin [0, N-1] : (x(n) = 0)$

Relationship between continuous-time FT and DFT

The goal of spectrum analysis is often to determine the frequency content of an analog (continuous-time) signal; very often, as in most modern spectrum analyzers, this is actually accomplished by sampling the analog signal, windowing (truncating) the data, and computing and plotting the magnitude of its DFT. It is thus essential to relate the DFT frequency samples back to the original analog frequency. Assuming that the analog signal is bandlimited and the sampling frequency exceeds twice that limit so that no frequency aliasing occurs, the relationship between the continuous-time Fourier frequency Ω (in radians) and the DTFT frequency ω imposed by sampling is $\omega = \Omega T$ where T is the sampling period. Through the relationship $\omega_k = \frac{2\pi k}{N}$ between the DTFT frequency ω and the DFT frequency index k, the correspondence between the DFT frequency index and the original analog frequency can be found:

$$\Omega = \frac{2\pi k}{NT}$$

or in terms of analog frequency f in Hertz (cycles per second rather than radians)

$$f = \frac{k}{NT}$$

for k in the range k between 0 and $\frac{N}{2}$. It is important to note that $k \in \left[\frac{N}{2} + 1, N - 1\right]$ correspond to **negative** frequencies due to the periodicity of the DTFT and the DFT.

Exercise:

Problem:

In general, will DFT frequency values $X(k)$ **exactly** equal samples of the analog Fourier transform X_a at the corresponding frequencies? That is, will $X(k) = X_a\left(\frac{2\pi k}{NT}\right)$?

Solution:

In general, **NO**. The DTFT exactly corresponds to the continuous-time Fourier transform only when the signal is bandlimited and sampled at more than twice its highest frequency. The DFT frequency values exactly correspond to frequency samples of the DTFT only when the discrete-time

signal is time-limited. However, a bandlimited continuous-time signal cannot be time-limited, so in general these conditions cannot both be satisfied.

It can, however, be true for a small class of analog signals which are not time-limited but happen to exactly equal zero at all **sample times** outside of the interval $n \in [0, N-1]$. The sinc function with a bandwidth equal to the Nyquist frequency and centered at $t = 0$ is an example.

Zero-Padding

If more than N equally spaced frequency samples of a length-N signal are desired, they can easily be obtained by **zero-padding** the discrete-time signal and computing a DFT of the longer length. In particular, if LN DTFT samples are desired of a length-N sequence, one can compute the length-LN DFT of a length-LN zero-padded sequence

$$z(n) = \begin{cases} x(n) & \text{if } 0 \leq n \leq N-1 \\ 0 & \text{if } N \leq n \leq LN-1 \end{cases}$$

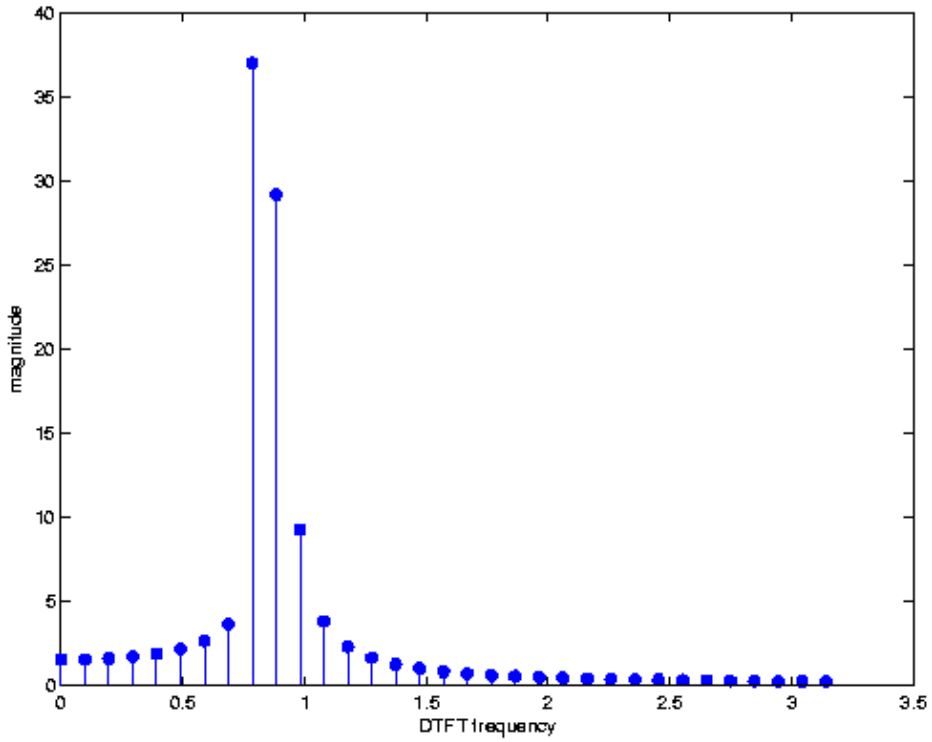
$$X\left(w_k = \frac{2\pi k}{LN}\right) = \sum_{n=0}^{N-1} x(n) e^{-\left(i\frac{2\pi kn}{LN}\right)} = \sum_{n=0}^{LN-1} z(n) e^{-\left(i\frac{2\pi kn}{LN}\right)} = \text{DFT}_{LN}\left[z[n]\right]$$

Note that zero-padding **interpolates** the spectrum. One should always zero-pad (by about at least a factor of 4) when using the DFT to approximate the DTFT to get a clear picture of the DTFT. While performing computations on zeros may at first seem inefficient, using FFT algorithms, which generally expect the same number of input and output samples, actually makes this approach very efficient.

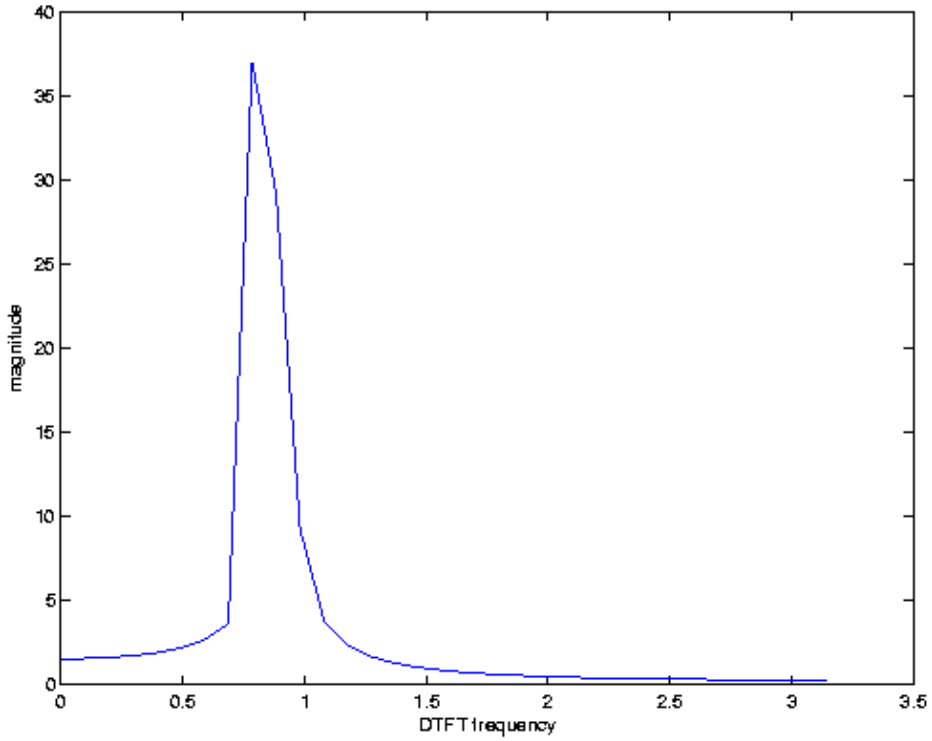
[link] shows the magnitude of the DFT values corresponding to the non-negative frequencies of a real-valued length-64 DFT of a length-64 signal, both in a "stem" format to emphasize the discrete nature of the DFT frequency samples, and as a line plot to emphasize its use as an approximation to the continuous-in-frequency DTFT. From this figure, it appears that the signal has a single dominant frequency component.
Spectrum without zero-padding

Magnitude DFT spectrum of 64 samples of a signal with a length-64 DFT (no zero padding)

Stem plot



Line Plot

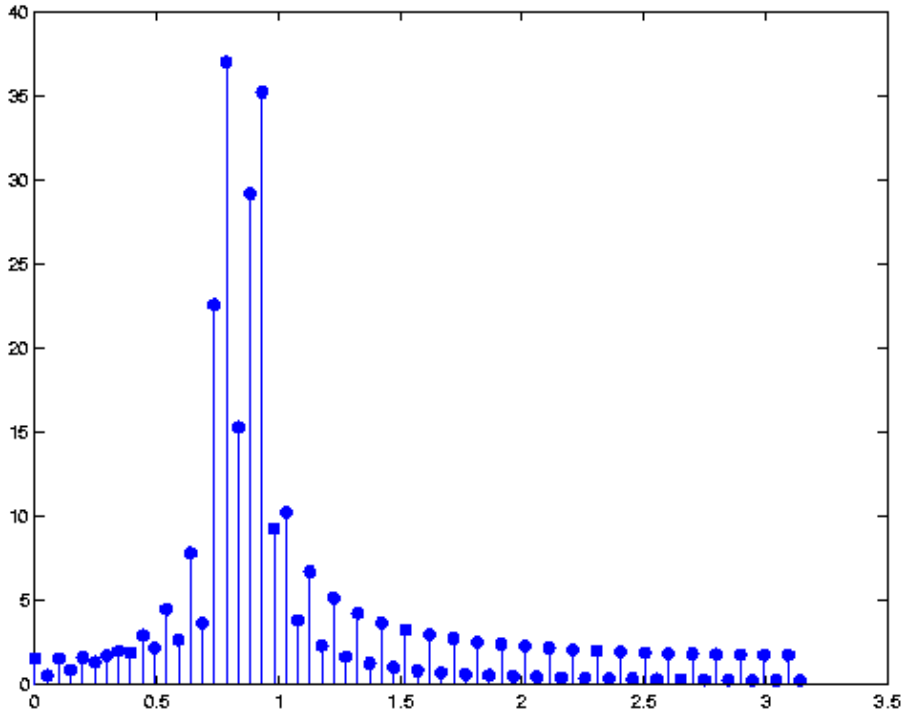


Zero-padding by a factor of two by appending 64 zero values to the signal and computing a length-128 DFT yields [link]. It can now be seen that the signal consists of at least two narrowband frequency components; the gap between them fell between DFT samples in [link], resulting in a misleading picture of the signal's spectral content. This is sometimes called the **picket-fence effect**, and is a result of insufficient sampling in frequency. While zero-padding by a factor of two has revealed more structure, it is unclear whether the peak magnitudes are reliably rendered, and the jagged linear interpolation in the line graph does not yet reflect the smooth, continuously-differentiable spectrum of the DTFT of a finite-length truncated signal. Errors in the apparent peak magnitude due to insufficient frequency sampling is sometimes referred to as **scalloping loss**.

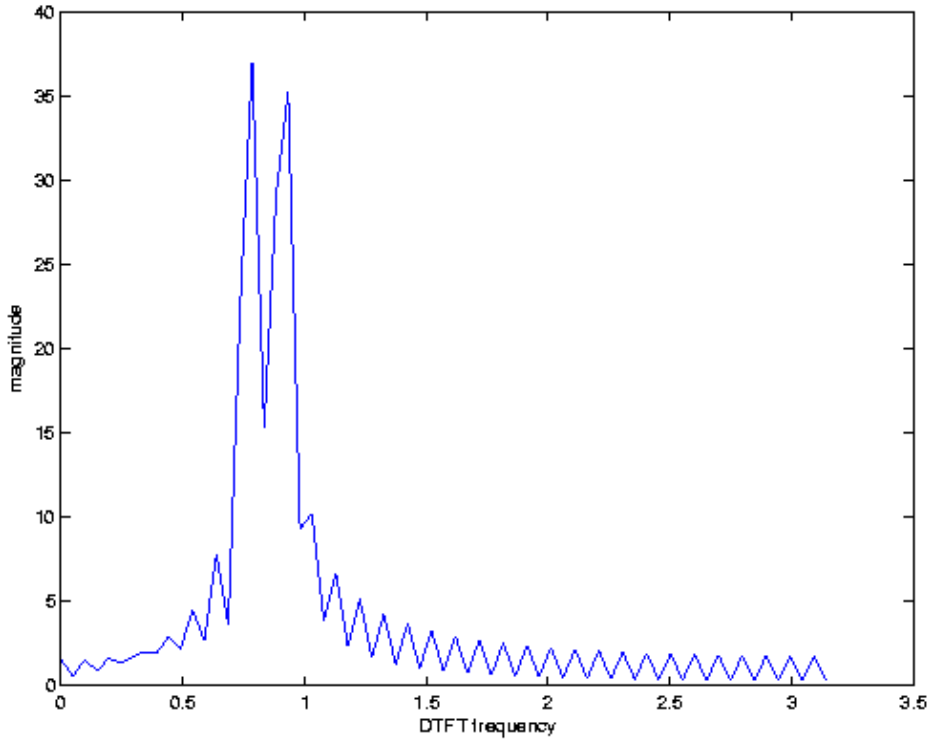
Spectrum with factor-of-two zero-padding

Magnitude DFT spectrum of 64 samples of a signal with a length-128 DFT (double-length zero-padding)

Stem plot


40
35
30
25
20
15
10
5
0
0
0.5
1
1.5
2
2.5
3
3.5

Line Plot

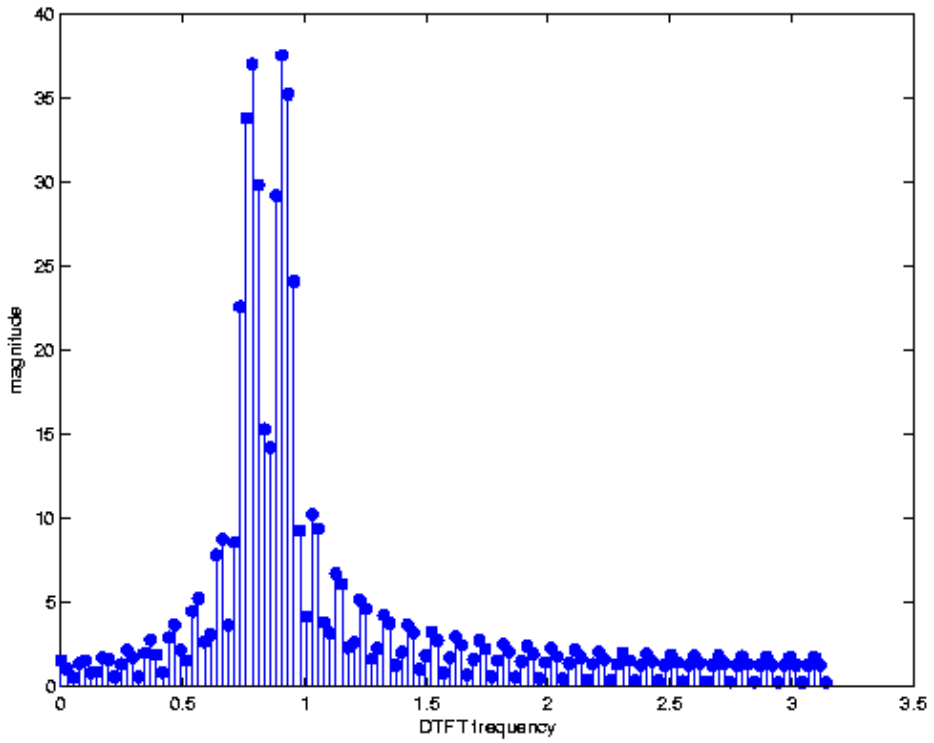

magnitude
DTFT frequency
40
35
30
25
20
15
10
5
0
0
0.5
1
1.5
2
2.5
3
3.5

Zero-padding to four times the length of the signal, as shown in [link], clearly shows the spectral structure and reveals that the magnitude of the two spectral lines are nearly identical. The line graph is still a bit rough and the peak magnitudes and frequencies may not be precisely captured, but the spectral characteristics of the truncated signal are now clear.

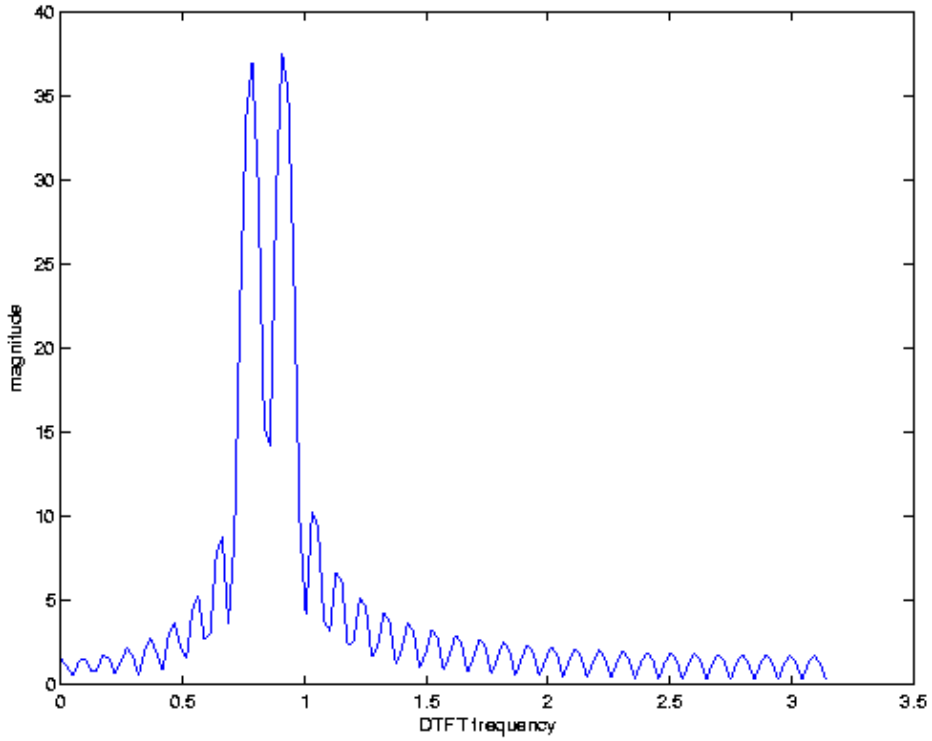
Spectrum with factor-of-four zero-padding

Magnitude DFT spectrum of 64 samples of a signal with a length-256 zero-padded DFT (four times zero-padding)

Stem plot



Line Plot

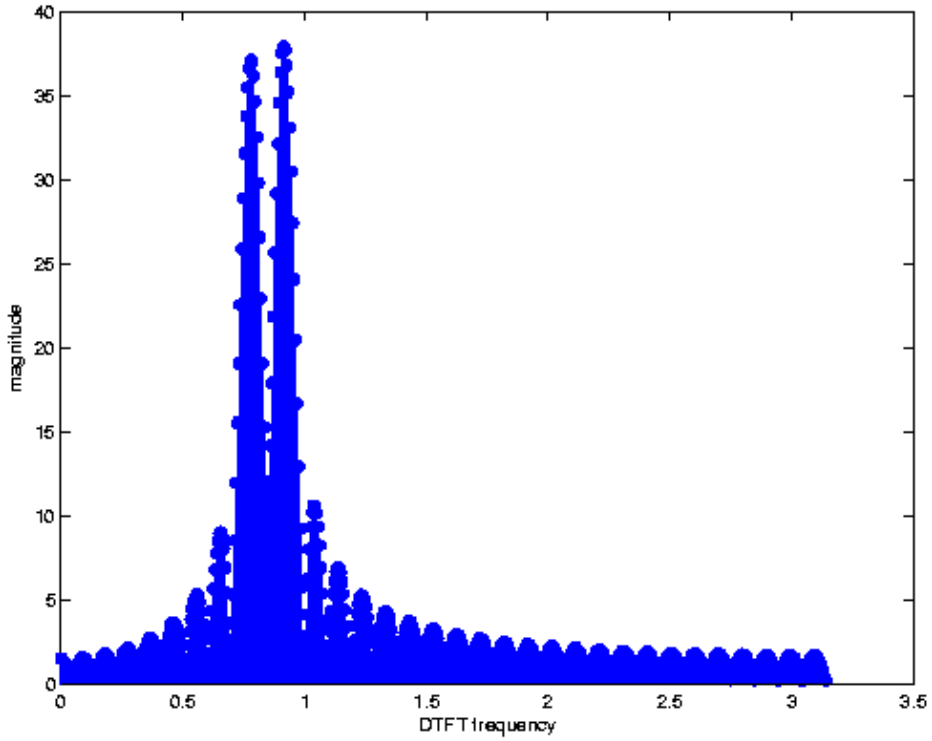


Zero-padding to a length of 1024, as shown in [link] yields a spectrum that is smooth and continuous to the resolution of the computer screen, and produces a very accurate rendition of the DTFT of the **truncated** signal.

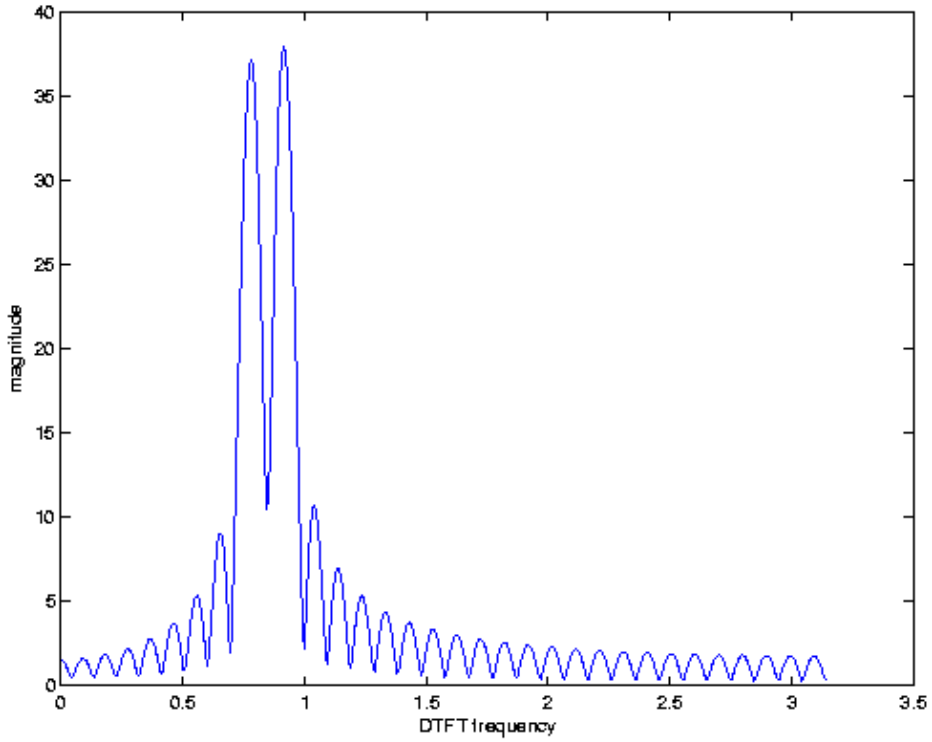
Spectrum with factor-of-sixteen zero-padding

Magnitude DFT spectrum of 64 samples of a signal with a length-1024 zero-padded DFT. The spectrum now looks smooth and continuous and reveals all the structure of the DTFT of a truncated signal.

Stem plot


magnitude
DTFT frequency
0
5
10
15
20
25
30
35
40
0
0.5
1
1.5
2
2.5
3
3.5

Line Plot


magnitude
DTFT frequency
0
5
10
15
20
25
30
35
40
0
0.5
1
1.5
2
2.5
3
3.5

The signal used in this example actually consisted of two pure sinusoids of equal magnitude. The slight difference in magnitude of the two dominant peaks, the breadth of the peaks, and the sinc-like lesser **side lobe** peaks throughout frequency are artifacts of the truncation, or windowing, process used to practically approximate the DFT. These problems and partial solutions to them are discussed in the following section.

Effects of Windowing

Applying the DTFT multiplication property

$$\widehat{X(\omega_k)} = \sum_{n=-\infty}^{\infty} x(n)w(n)e^{-(i\omega_k n)} = \frac{1}{2\pi}X(\omega_k)*W(\omega_k)$$

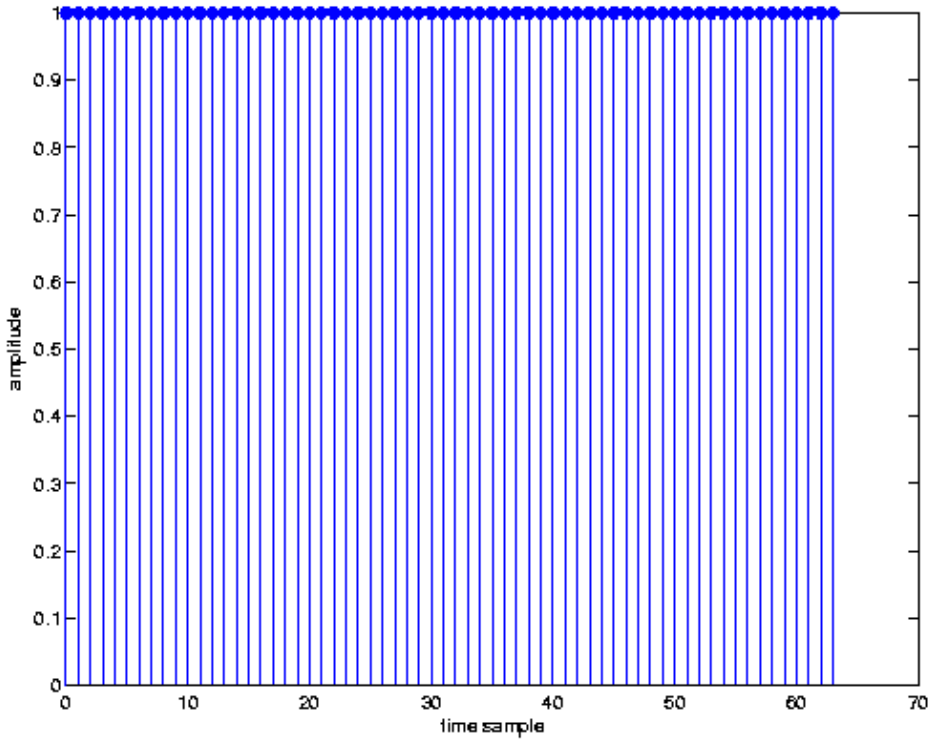
we find that the DFT of the windowed (truncated) signal produces samples not of the true (desired) DTFT spectrum $X(\omega)$, but of a **smoothed** verson $X(\omega)*W(\omega)$. We want this to resemble $X(\omega)$ as closely as possible, so $W(\omega)$ should be as close to an impulse as possible. The **window** $w(n)$ need not be a simple **truncation** (or **rectangle**, or **boxcar**) window; other shapes can also be used as long as they limit the sequence to at most N consecutive nonzero samples. All good windows are impulse-like, and represent various tradeoffs between three criteria:

1. main lobe width: (limits resolution of closely-spaced peaks of equal height)
2. height of first sidelobe: (limits ability to see a small peak near a big peak)
3. slope of sidelobe drop-off: (limits ability to see small peaks further away from a big peak)

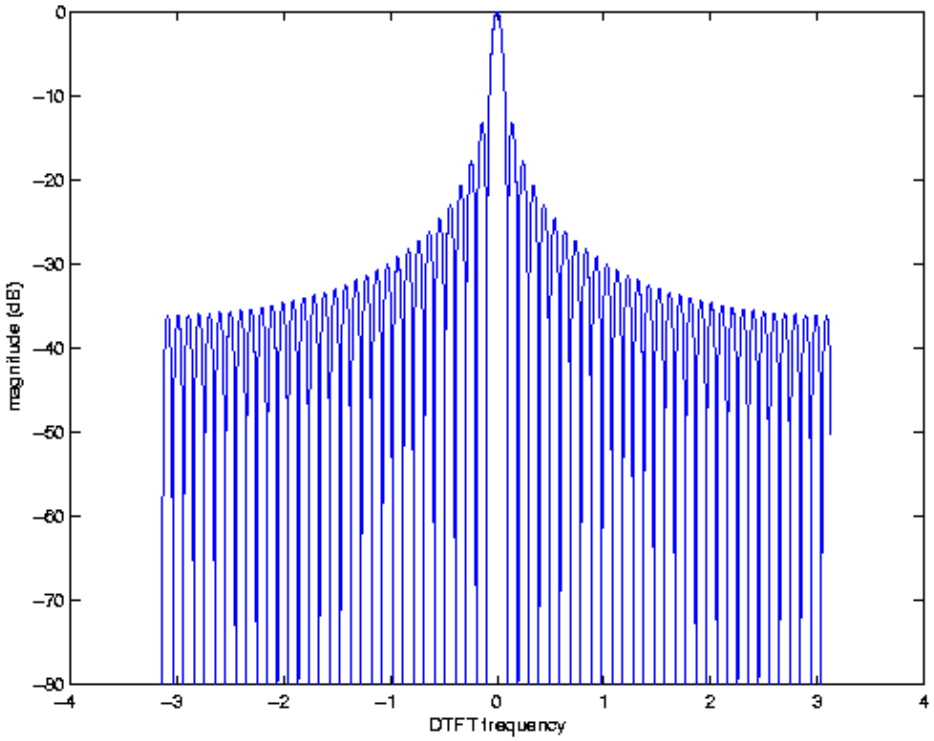
Many different window functions have been developed for truncating and shaping a length-N signal segment for spectral analysis. The simple **truncation** window has a periodic sinc DTFT, as shown in [link]. It has the narrowest main-lobe width, $\frac{2\pi}{N}$ at the -3 dB level and $\frac{4\pi}{N}$ between the two zeros surrounding the main lobe, of the common window functions, but also the largest side-lobe peak, at about -13 dB. The side-lobes also taper off relatively slowly.

Length-64 truncation (boxcar) window and its magnitude DFT spectrum

Rectangular window



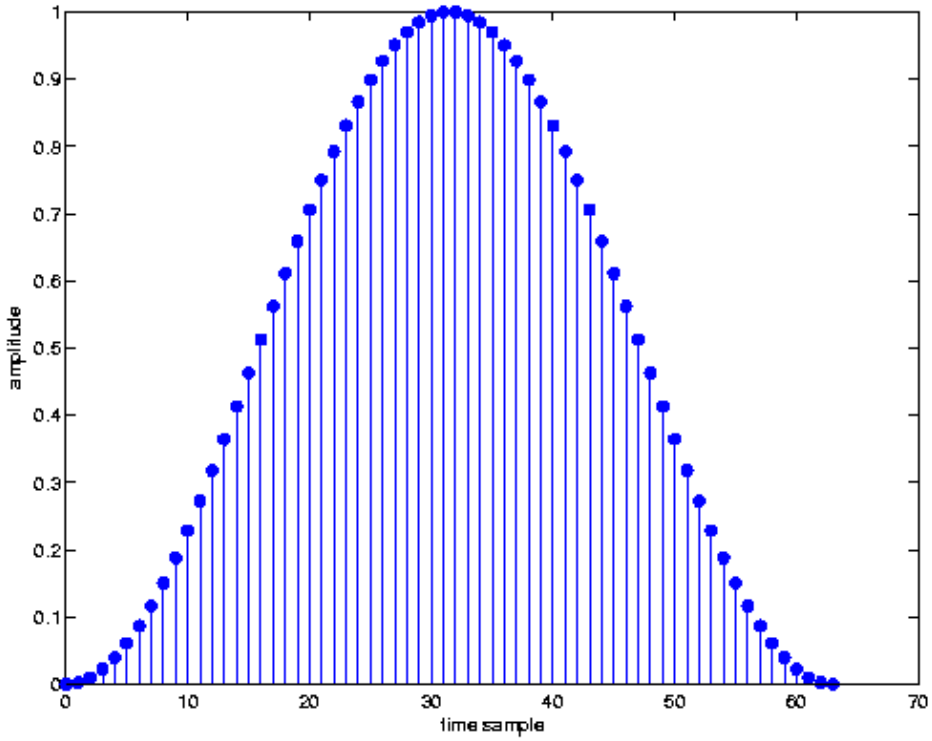
Magnitude of boxcar window spectrum



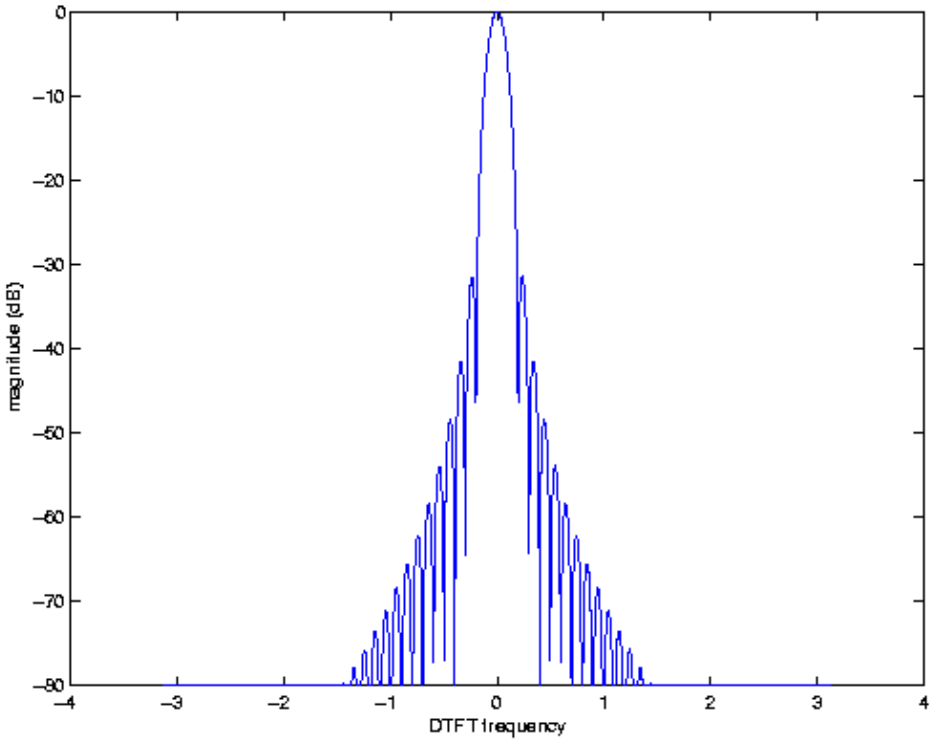
The **Hann window** (sometimes also called the **hanning** window), illustrated in [link], takes the form $w[n] = 0.5 - 0.5\cos\left(\frac{2\pi n}{N-1}\right)$ for n between 0 and $N-1$. It has a main-lobe width (about $\frac{3\pi}{N}$ at the -3 dB level and $\frac{8\pi}{N}$ between the two zeros surrounding the main lobe) considerably larger than the rectangular window, but the largest side-lobe peak is much lower, at about -31.5 dB. The side-lobes also taper off much faster. For a given length, this window is worse than the boxcar window at separating closely-spaced spectral components of similar magnitude, but better for identifying smaller-magnitude components at a greater distance from the larger components.

Length-64 Hann window and its magnitude DFT spectrum

Hann window



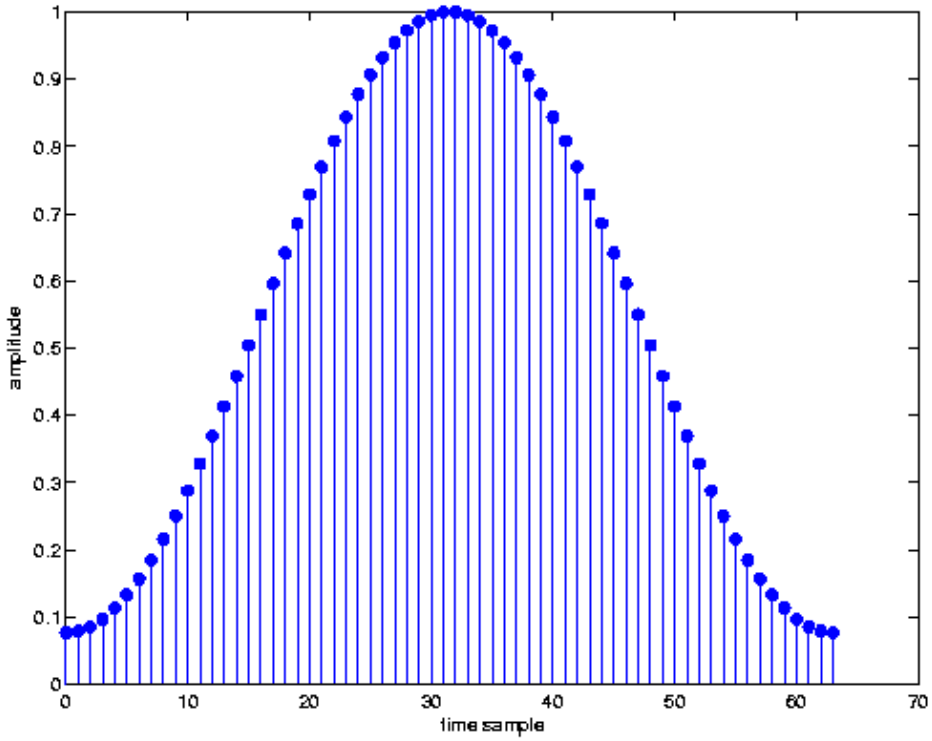
Magnitude of Hann window spectrum



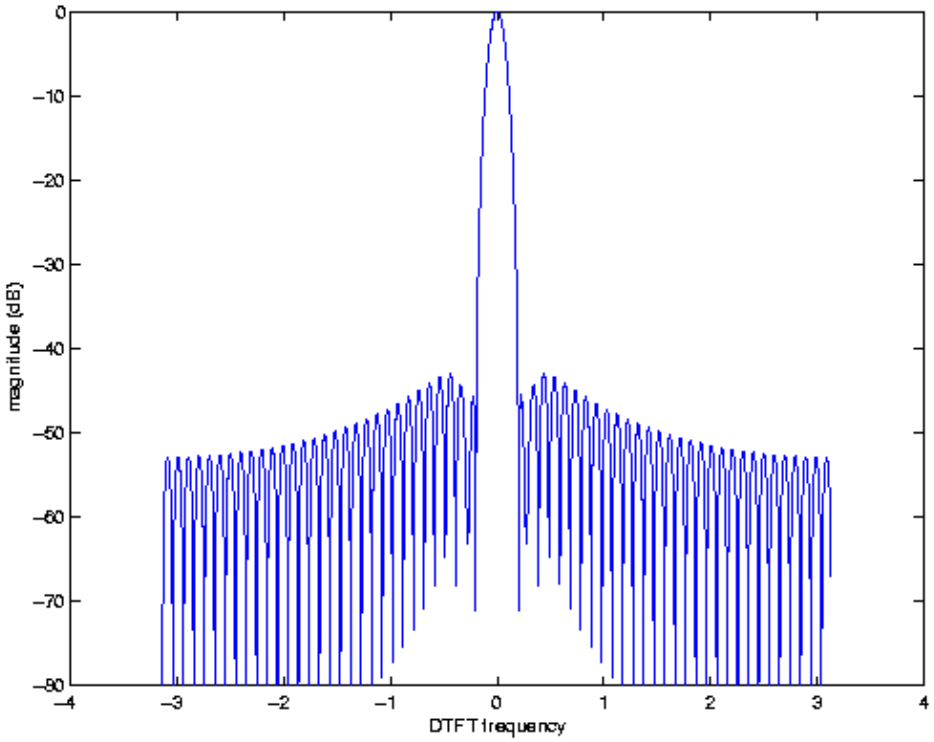
The **Hamming window**, illustrated in [link], has a form similar to the Hann window but with slightly different constants:
$w[n] = 0.538 - 0.462\cos\left(\frac{2\pi n}{N-1}\right)$ for n between 0 and $N-1$. Since it is composed of the same Fourier series harmonics as the Hann window, it has a similar main-lobe width (a bit less than $\frac{3\pi}{N}$ at the -3 dB level and $\frac{8\pi}{N}$ between the two zeros surrounding the main lobe), but the largest side-lobe peak is much lower, at about -42.5 dB. However, the side-lobes also taper off much more slowly than with the Hann window. For a given length, the Hamming window is better than the Hann (and of course the boxcar) windows at separating a small component relatively near to a large component, but worse than the Hann for identifying very small components at considerable frequency separation. Due to their shape and form, the Hann and Hamming windows are also known as **raised-cosine windows**.

Length-64 Hamming window and its magnitude DFT spectrum

Hamming window



Magnitude of Hamming window spectrum

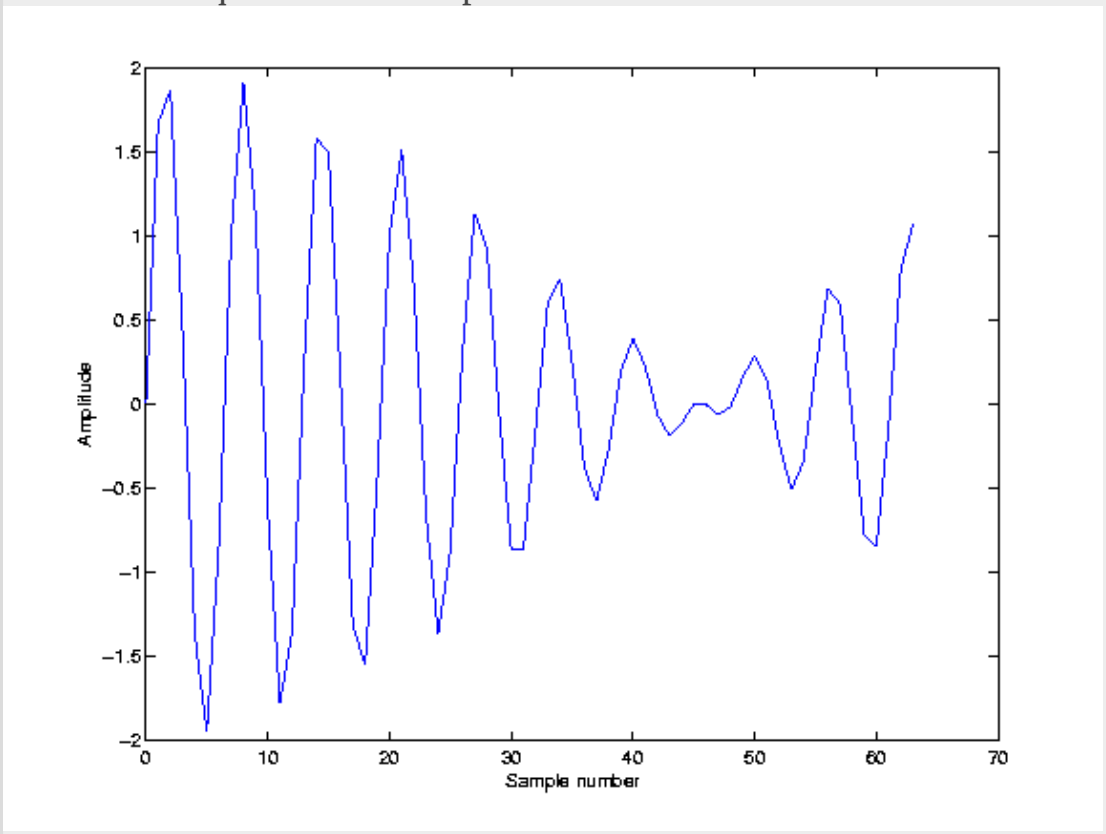


Note: Standard even-length windows are symmetric around a point halfway between the window samples $\frac{N}{2} - 1$ and $\frac{N}{2}$. For some applications such as time-frequency analysis, it may be important to align the window perfectly to a sample. In such cases, a **DFT-symmetric** window that is symmetric around the $\frac{N}{2}$-th sample can be used. For example, the DFT-symmetric Hamming window is $w[n] = 0.538 - 0.462\cos\left(\frac{2\pi n}{N}\right)$. A DFT-symmetric window has a purely real-valued DFT and DTFT. DFT-symmetric versions of windows, such as the Hamming and Hann windows, composed of few discrete Fourier series terms of period N, have few non-zero DFT terms (only when **not** zero-padded) and can be used efficiently in running FFTs.

The main-lobe width of a window is an inverse function of the window-length N; for any type of window, a longer window will always provide better resolution.

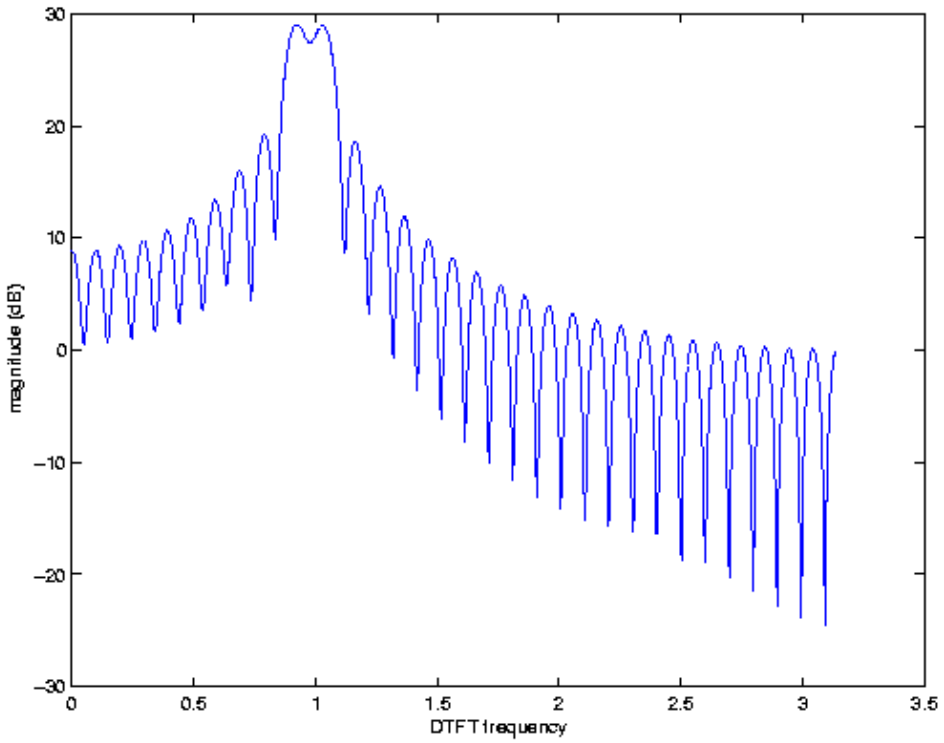
Many other windows exist that make various other tradeoffs between main-lobe width, height of largest side-lobe, and side-lobe rolloff rate. The Kaiser window family, based on a modified Bessel function, has an adjustable parameter that allows the user to tune the tradeoff over a continuous range. The Kaiser window has near-optimal time-frequency resolution and is widely used. A list of many different windows can be found here.

Example:
[link] shows 64 samples of a real-valued signal composed of several sinusoids of various frequencies and amplitudes.



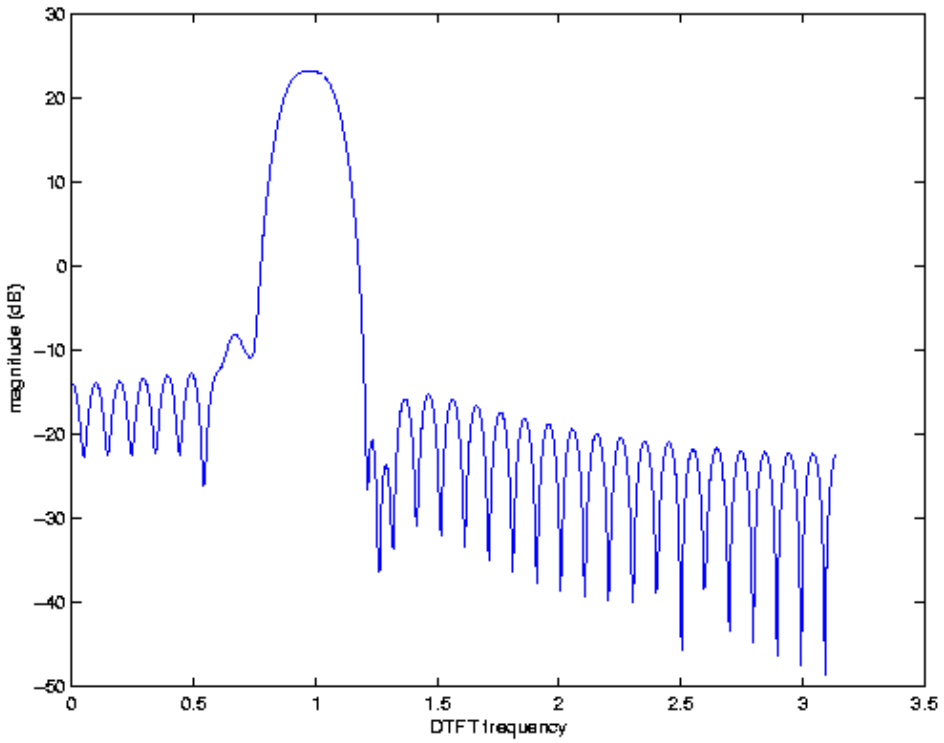
64 samples of an unknown signal

[link] shows the magnitude (in dB) of the positive frequencies of a length-1024 zero-padded DFT of this signal (that is, using a simple truncation, or rectangular, window).



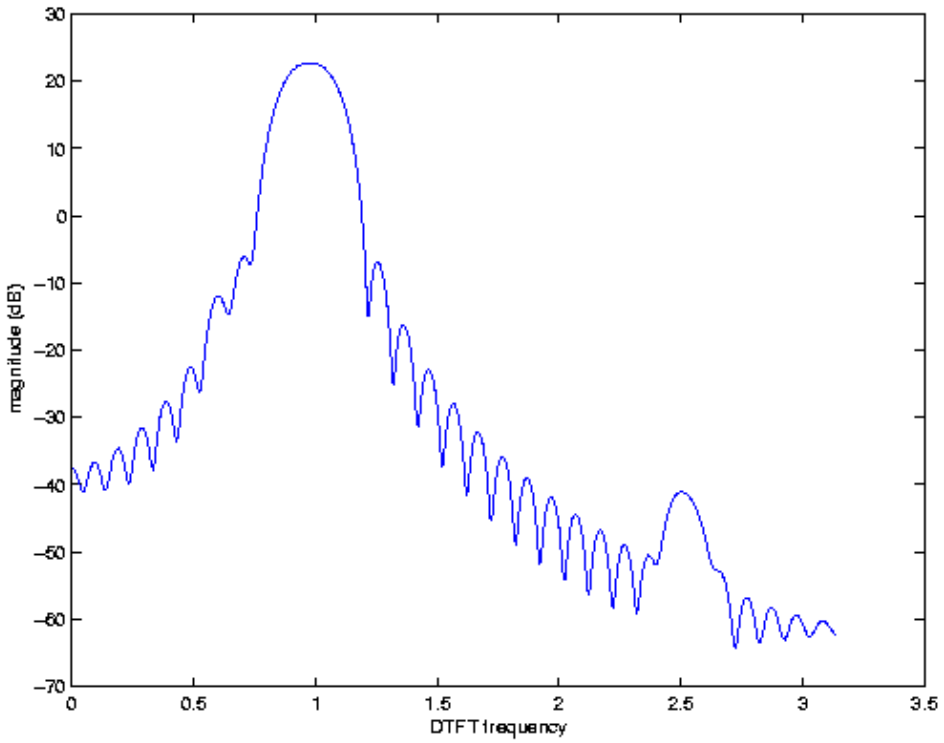
Magnitude (in dB) of the zero-padded DFT spectrum of the signal in [link] using a simple length-64 rectangular window

From this spectrum, it is clear that the signal has two large, nearby frequency components with frequencies near 1 radian of essentially the same magnitude. [link] shows the spectral estimate produced using a length-64 Hamming window applied to the same signal shown in [link].



Magnitude (in dB) of the zero-padded DFT spectrum of the signal in [link] using a length-64 Hamming window

The two large spectral peaks can no longer be resolved; they blur into a single broad peak due to the reduced spectral resolution of the broader main lobe of the Hamming window. However, the lower side-lobes reveal a third component at a frequency of about 0.7 radians at about 35 dB lower magnitude than the larger components. This component was entirely buried under the side-lobes when the rectangular window was used, but now stands out well above the much lower nearby side-lobes of the Hamming window.
[link] shows the spectral estimate produced using a length-64 Hann window applied to the same signal shown in [link].



Magnitude (in dB) of the zero-padded DFT spectrum of the signal in [link] using a length-64 Hann window

The two large components again merge into a single peak, and the smaller component observed with the Hamming window is largely lost under the higher nearby side-lobes of the Hann window. However, due to the much faster side-lobe rolloff of the Hann window's spectrum, a fourth component at a frequency of about 2.5 radians with a magnitude about 65 dB below that of the main peaks is now clearly visible.

This example illustrates that no single window is best for all spectrum analyses. The best window depends on the nature of the signal, and different windows may be better for different components of the same signal. A skilled spectrum analysist may apply several different windows to a signal to gain a fuller understanding of the data.

Classical Statistical Spectral Estimation

Many signals are either partly or wholly stochastic, or random. Important examples include human speech, vibration in machines, and CDMA communication signals. Given the ever-present noise in electronic systems, it can be argued that almost **all** signals are at least partly stochastic. Such signals may have a distinct **average** spectral structure that reveals important information (such as for speech recognition or early detection of damage in machinery). Spectrum analysis of any single block of data using window-based deterministic spectrum analysis, however, produces a random spectrum that may be difficult to interpret. For such situations, the classical statistical spectrum estimation methods described in this module can be used.

The goal in classical statistical spectrum analysis is to estimate $E\left[\left(|X(\omega)|\right)^2\right]$, the **power spectral density (PSD)** across frequency of the stochastic signal. That is, the goal is to find the expected (mean, or average) energy density of the signal as a function of frequency. (For zero-mean signals, this equals the variance of each frequency sample.) Since the spectrum of each block of signal samples is itself random, we must average the squared spectral magnitudes over a number of blocks of data to find the mean. There are two main classical approaches, the periodogram and auto-correlation methods.

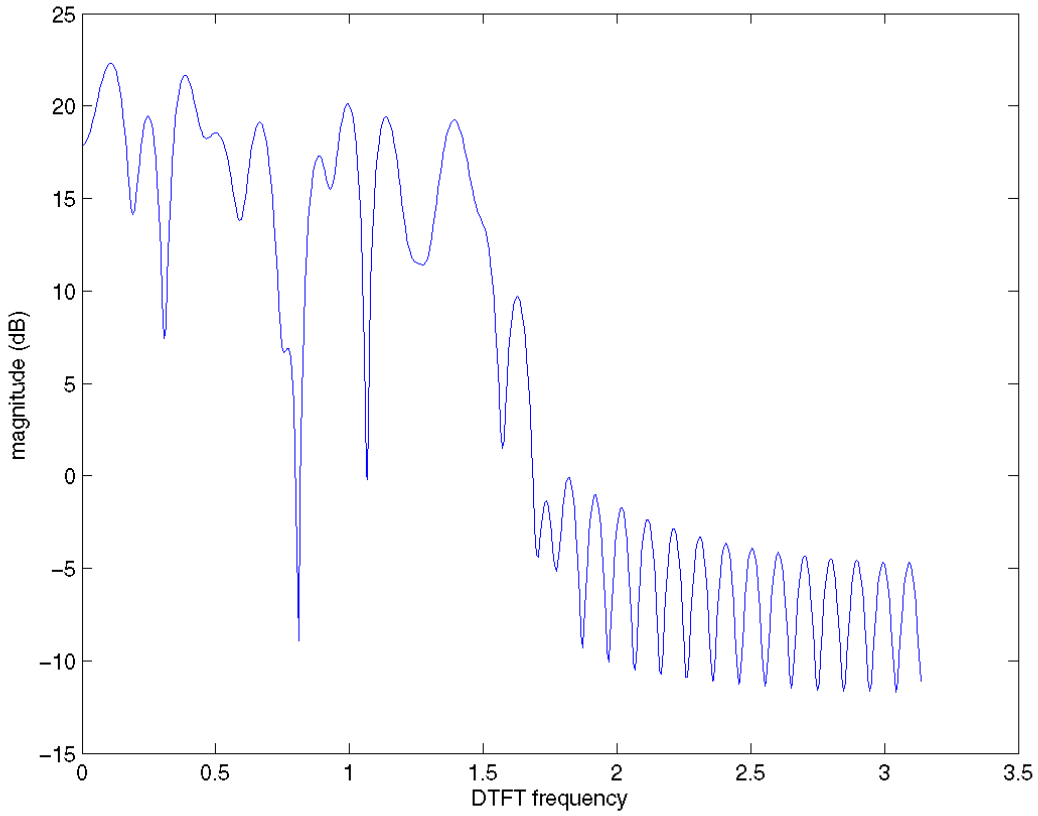
Periodogram method

The periodogram method divides the signal into a number of shorter (and often overlapped) blocks of data, computes the squared magnitude of the windowed (and usually zero-padded) DFT, $X_i(\omega_k)$, of each block, and averages them to estimate the power spectral density. The squared magnitudes of the DFTs of L possibly overlapped length-N windowed blocks of signal (each probably with zero-padding) are averaged to estimate the power spectral density:

$$\widehat{X(\omega_k)} = \frac{1}{L}\sum_{i=1}^{L}\left(|X_i(\omega_k)|\right)^2$$

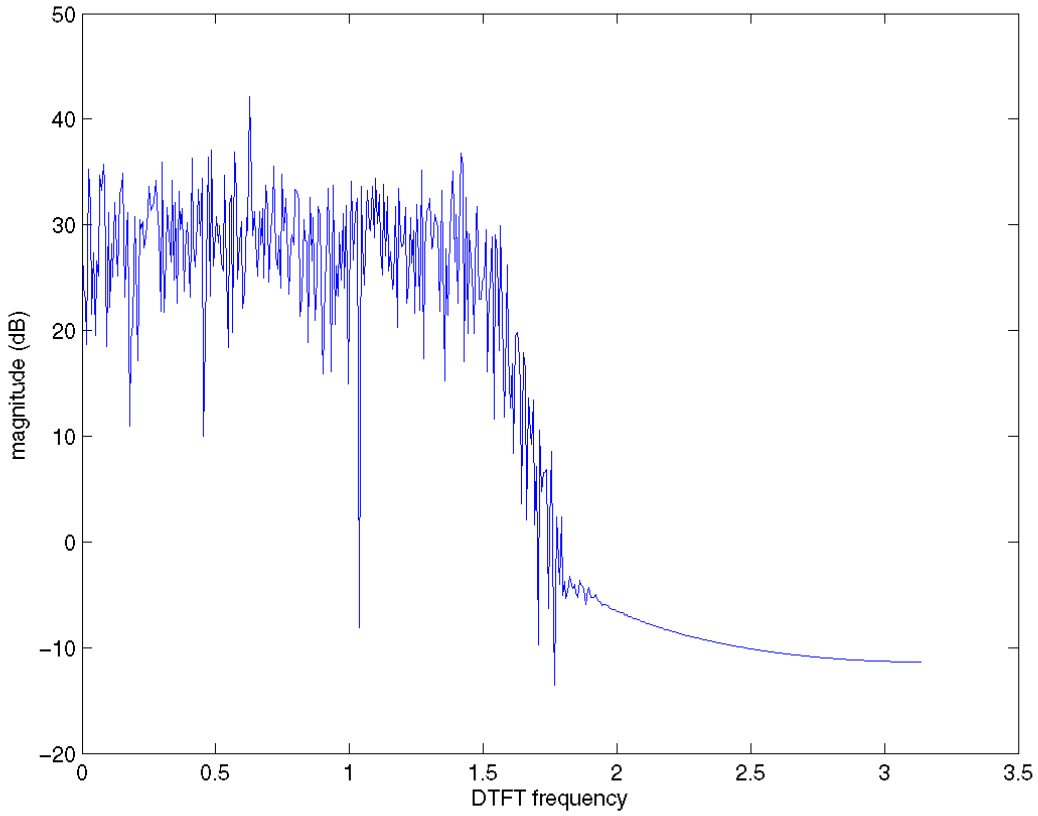
For a fixed **total** number of samples, this introduces a tradeoff: Larger individual data blocks provides better frequency resolution due to the use of a longer window, but it means there are less blocks to average, so the estimate has higher variance and appears more noisy. The best tradeoff depends on the application. Overlapping blocks by a factor of two to four increases the number of averages and reduces the variance, but since the same data is being reused, still more overlapping does not further reduce the variance. As with any window-based spectrum estimation procedure, the window function introduces broadening and sidelobes into the power spectrum estimate. That is, the periodogram produces an estimate of the **windowed** spectrum $\widehat{X(\omega)} = E\left[\left(|X(\omega)*W_M|\right)^2\right]$, not of $E\left[\left(|X(\omega)|\right)^2\right]$.

Example:
[link] shows the non-negative frequencies of the DFT (zero-padded to 1024 total samples) of 64 samples of a real-valued stochastic signal.



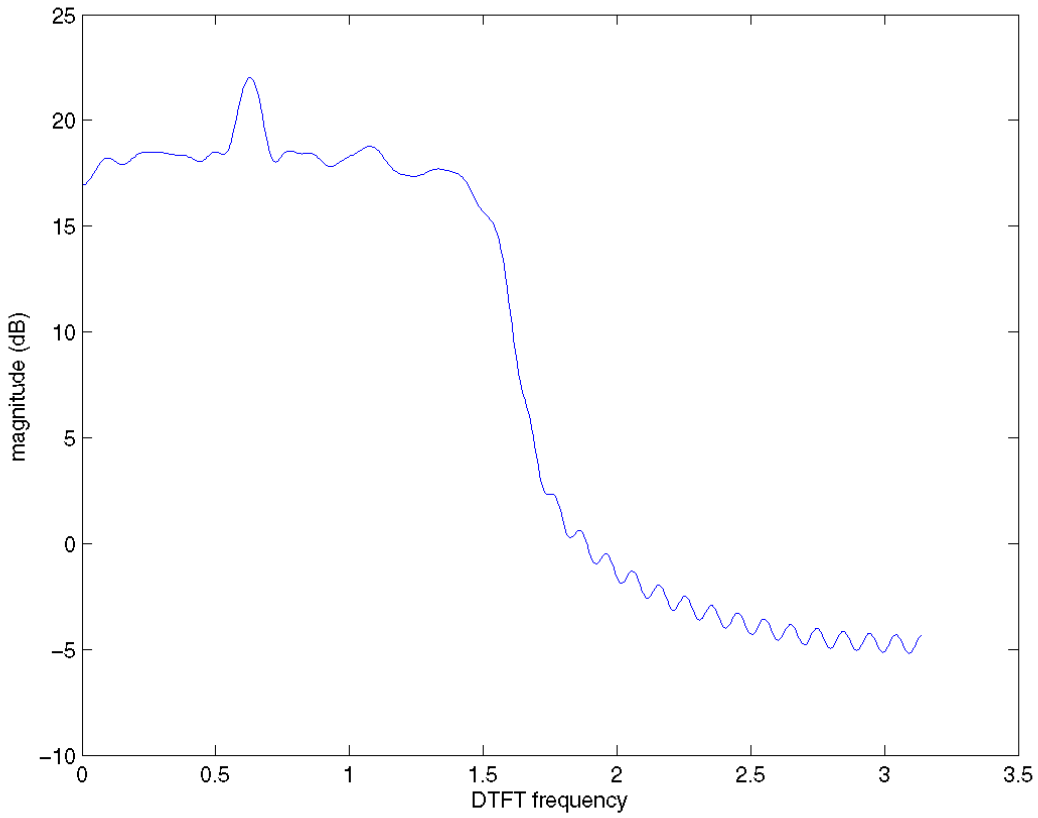
DFT magnitude (in dB) of 64 samples of a stochastic signal

With no averaging, the power spectrum is very noisy and difficult to interpret other than noting a significant reduction in spectral energy above about half the Nyquist frequency. Various peaks and valleys appear in the lower frequencies, but it is impossible to say from this figure whether they represent actual structure in the power spectral density (PSD) or simply random variation in this single realization. [link] shows the same frequencies of a length-1024 DFT of a length-1024 signal. While the frequency resolution has improved, there is still no averaging, so it remains difficult to understand the power spectral density of this signal. Certain small peaks in frequency might represent narrowband components in the spectrum, or may just be random noise peaks.



DFT magnitude (in dB) of 1024 samples of a stochastic signal

In [link], a power spectral density computed from averaging the squared magnitudes of length-1024 zero-padded DFTs of 508 length-64 blocks of data (overlapped by a factor of four, or a 16-sample step between blocks) are shown.



Power spectrum density estimate (in dB) of 1024 samples of a stochastic signal

While the frequency resolution corresponds to that of a length-64 truncation window, the averaging greatly reduces the variance of the spectral estimate and allows the user to reliably conclude that the signal consists of lowpass broadband noise with a flat power spectrum up to half the Nyquist frequency, with a stronger narrowband frequency component at around 0.65 radians.

Auto-correlation-based approach

The averaging necessary to estimate a power spectral density can be performed in the discrete-time domain, rather than in frequency, using the auto-correlation method. The squared magnitude of the frequency response, from the DTFT multiplication and conjugation properties, corresponds in the discrete-time domain to the signal convolved with the time-reverse of itself,

$$\left(|X(\omega)|\right)^2 = X(\omega)X^*(\omega) \leftrightarrow \Big(x(n), x^*(-n)\Big) = r(n)$$

or its **auto-correlation**

$$r(n) = \sum_k x(k)x^*(n+k)$$

We can thus compute the squared magnitude of the spectrum of a signal by computing the DFT of its auto-correlation. For stochastic signals, the power spectral density is an expectation, or average, and by linearity of expectation can be found by transforming the average of the auto-correlation. For a finite block of N signal samples, the average of the autocorrelation values, $r(n)$, is

$$r(n) = \frac{1}{N-n} \sum_{k=0}^{N-(1-n)} x(k)x^*(n+k)$$

Note that with increasing **lag**, n, fewer values are averaged, so they introduce more noise into the estimated power spectrum. By windowing the auto-correlation before transforming it to the frequency domain, a less noisy power spectrum is obtained, at the expense of less resolution. The multiplication property of the DTFT shows that the windowing smooths the resulting power spectrum via convolution with the DTFT of the window:

$$\widehat{X(\omega)} = \sum_{n=-M}^{M} r(n)w(n)e^{-(i\omega n)} = \Big(E\Big[\left(|X(\omega)|\right)^2\Big]\Big) * W(\omega)$$

This yields another important interpretation of how the auto-correlation method works: it estimates the power spectral density by **averaging the power spectrum over nearby frequencies**, through convolution with the window function's transform, to reduce variance. Just as with the periodogram approach, there is always a variance vs. resolution tradeoff. The periodogram and the auto-correlation method give similar results for a similar amount of averaging; the user should simply note that in the periodogram case, the window introduces smoothing of the spectrum via frequency convolution **before** squaring the magnitude, whereas the periodogram convolves the squared magnitude with $W(\omega)$.

Short Time Fourier Transform

Short Time Fourier Transform

The Fourier transforms (FT, DTFT, DFT, etc.) do not clearly indicate how the frequency content of a signal changes over time.

That information is hidden in the phase - it is not revealed by the plot of the magnitude of the spectrum.

> **Note:** To see how the frequency content of a signal changes over time, we can cut the signal into blocks and compute the spectrum of each block.

To improve the result,

1. blocks are overlapping
2. each block is multiplied by a window that is tapered at its endpoints.

Several parameters must be chosen:

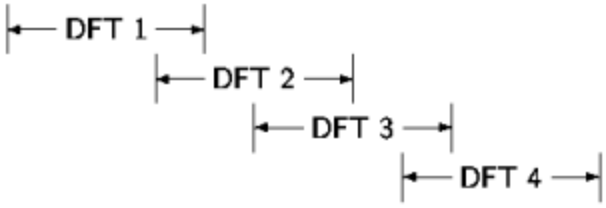
- Block length, R.
- The type of window.
- Amount of overlap between blocks. ([link])
- Amount of zero padding, if any.

STFT: Overlap Parameter

NO OVERLAP



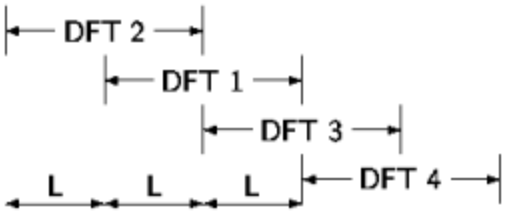
R/4 OVERLAP



R/2 OVERLAP



The parameter L



L is the number of samples between adjacent blocks.

The short-time Fourier transform is defined as
Equation:

$$\begin{aligned} X(\omega, m) &= \operatorname{STFT}(x(n)) \coloneqq \operatorname{DTFT}(x(n-m)w(n)) \\ &= \sum_{n=-\infty}^{\infty} x(n-m)w(n)e^{-(i\omega n)} \\ &= \sum_{n=0}^{R-1} x(n-m)w(n)e^{-(i\omega n)} \end{aligned}$$

where $w(n)$ is the window function of length R.

1. The STFT of a signal $x(n)$ is a function of two variables: time and frequency.
2. The block length is determined by the support of the window function $w(n)$.
3. A graphical display of the magnitude of the STFT, $|X(\omega, m)|$, is called the **spectrogram** of the signal. It is often used in speech processing.
4. The STFT of a signal is invertible.
5. One can choose the block length. A long block length will provide higher frequency resolution (because the main-lobe of the window function will be narrow). A short block length will provide higher time resolution because less averaging across samples is performed for each STFT value.
6. A **narrow-band spectrogram** is one computed using a relatively long block length R, (long window function).
7. A **wide-band spectrogram** is one computed using a relatively short block length R, (short window function).

Sampled STFT

To numerically evaluate the STFT, we sample the frequency axis ω in N equally spaced samples from $\omega = 0$ to $\omega = 2\pi$.
Equation:

$$\forall k, 0 \leq k \leq N-1 : \left(\omega_k = \frac{2\pi}{N} k\right)$$

We then have the discrete STFT,
Equation:

$$\begin{aligned} X^d(k,m) \coloneqq X\left(\tfrac{2\pi}{N}k, m\right) &= \textstyle\sum_{n=0}^{R-1} x(n-m)w(n)e^{-(i\omega n)} \\ &= \textstyle\sum_{n=0}^{R-1} x(n-m)w(n){W_N}^{-(kn)} \\ &= \mathrm{DFT}_N\left(x(n-m)w(n)|_{n=0}^{R-1}, 0, \ldots 0\right) \end{aligned}$$

where $0,\ldots 0$ is $N-R$.

In this definition, the overlap between adjacent blocks is $R-1$. The signal is shifted along the window one sample at a time. That generates more points than is usually needed, so we also sample the STFT along the time direction. That means we usually evaluate

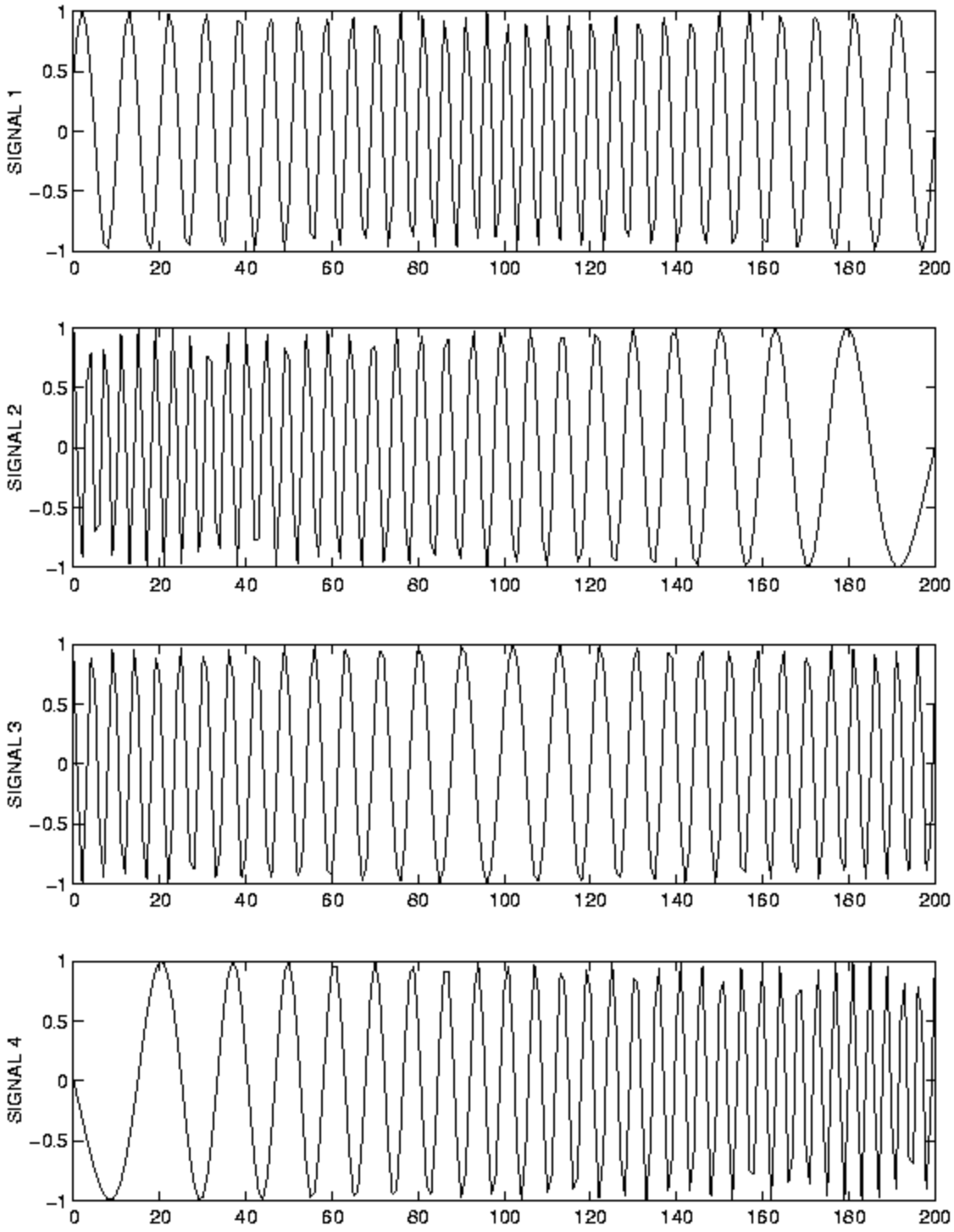
$$X^d(k, Lm)$$

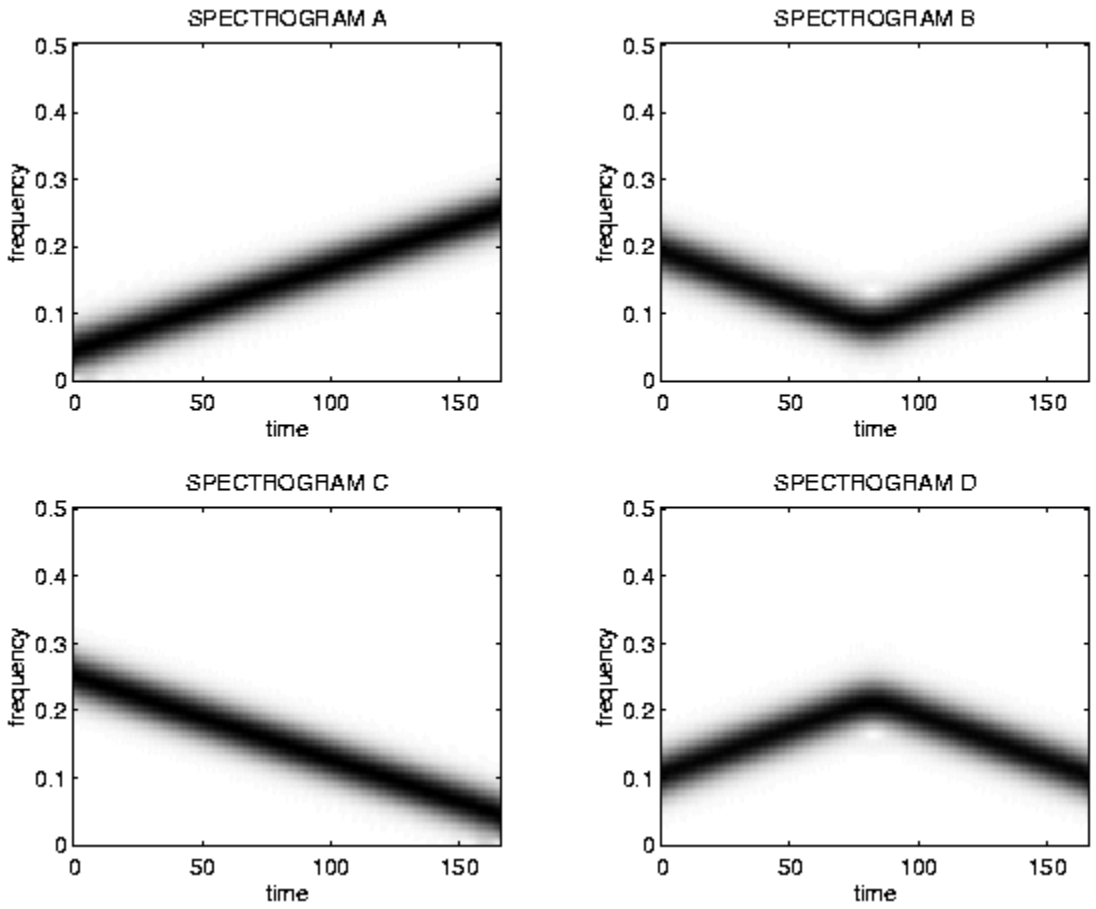
where L is the time-skip. The relation between the time-skip, the number of overlapping samples, and the block length is

$$\text{Overlap} = R - L$$

Exercise:

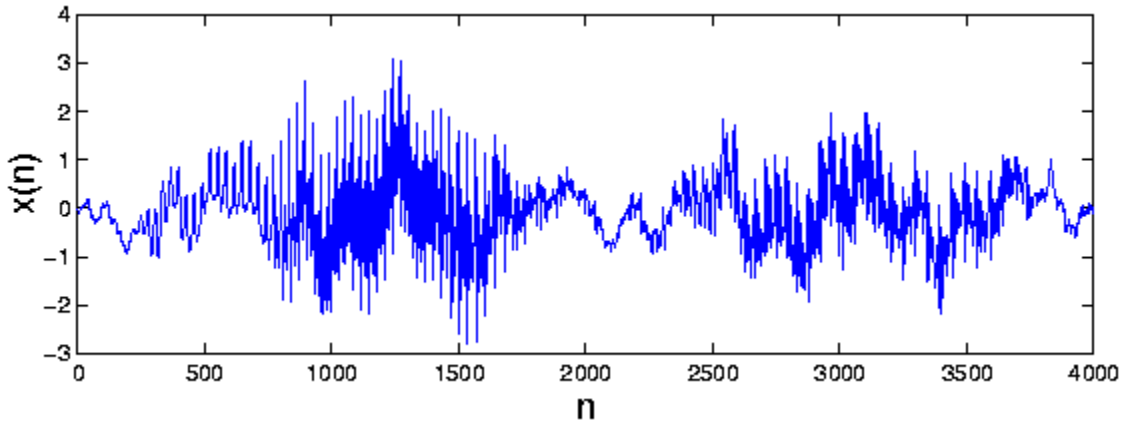
Problem: Match each signal to its spectrogram in [link].

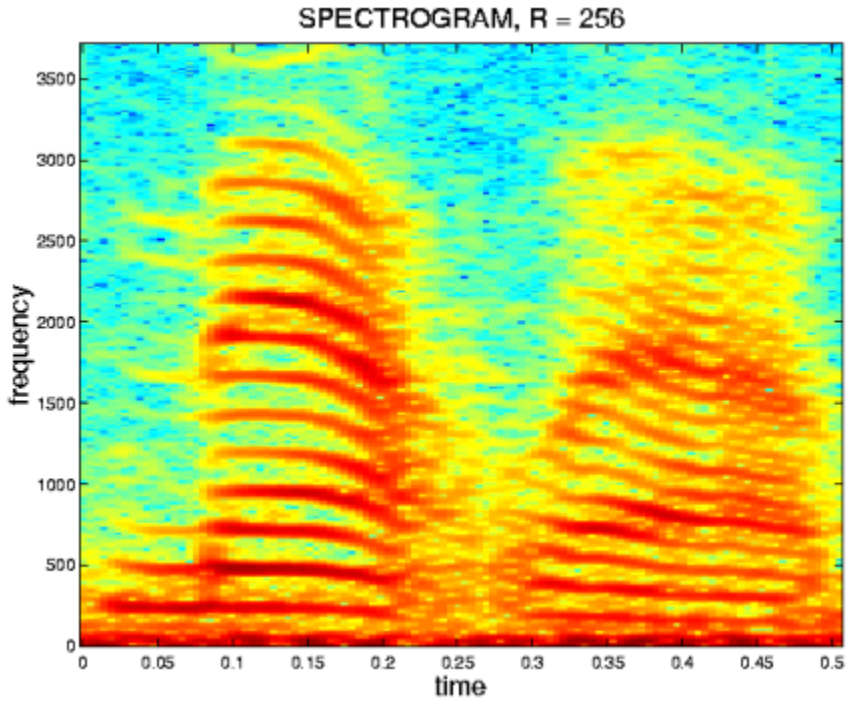

SIGNAL 1
SIGNAL 2
SIGNAL 3
SIGNAL 4
1
0.5
0
−0.5
−1
0
20
40
60
80
100
120
140
160
180
200



Solution:

Spectrogram Example





The matlab program for producing the figures above ([link] and [link]).

```
% LOAD DATA
load mtlb;
x = mtlb;
```

```
figure(1), clf
plot(0:4000,x)
xlabel('n')
ylabel('x(n)')

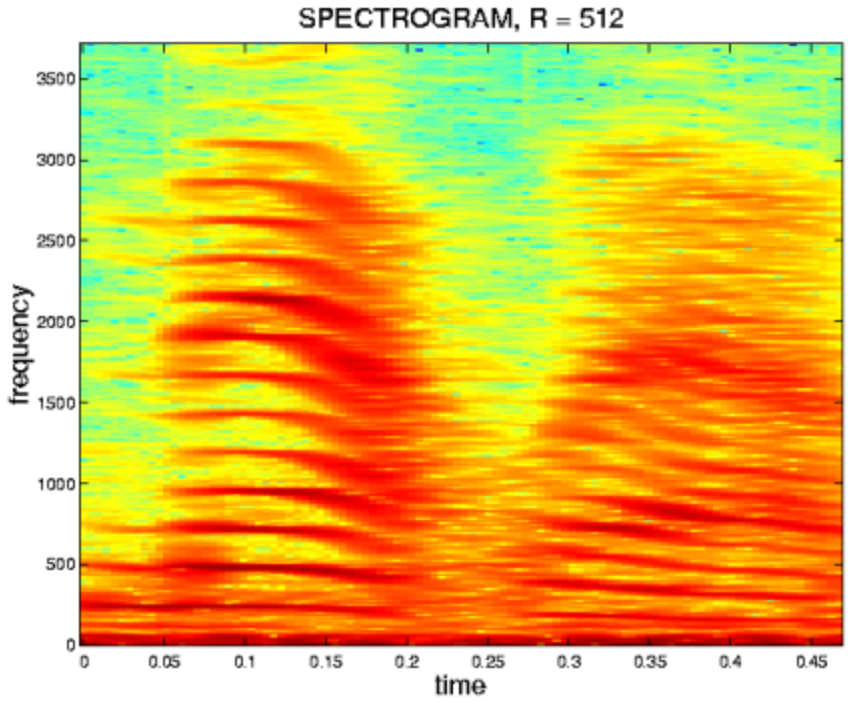
% SET PARAMETERS
R = 256;                % R: block length
window = hamming(R);    % window function
of length R
N = 512;                % N: frequency
discretization
L = 35;                 % L: time lapse
between blocks
fs = 7418;              % fs: sampling
frequency
overlap = R - L;

% COMPUTE SPECTROGRAM
[B,f,t] =
specgram(x,N,fs,window,overlap);

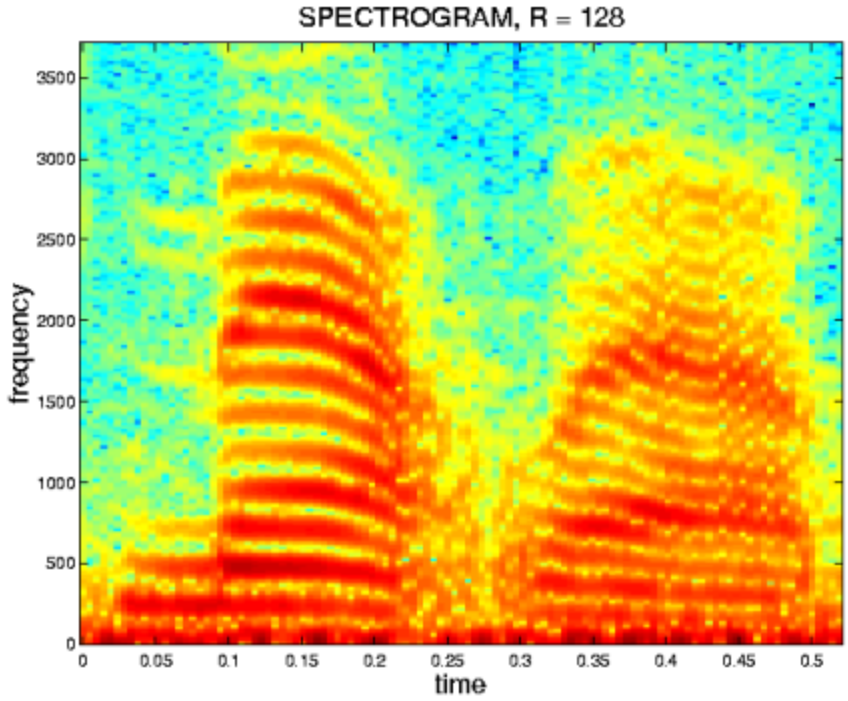
% MAKE PLOT
figure(2), clf
imagesc(t,f,log10(abs(B)));
colormap('jet')
axis xy
xlabel('time')
ylabel('frequency')
title('SPECTROGRAM, R = 256')
```

Effect of window length R

Narrow-band spectrogram: better frequency resolution

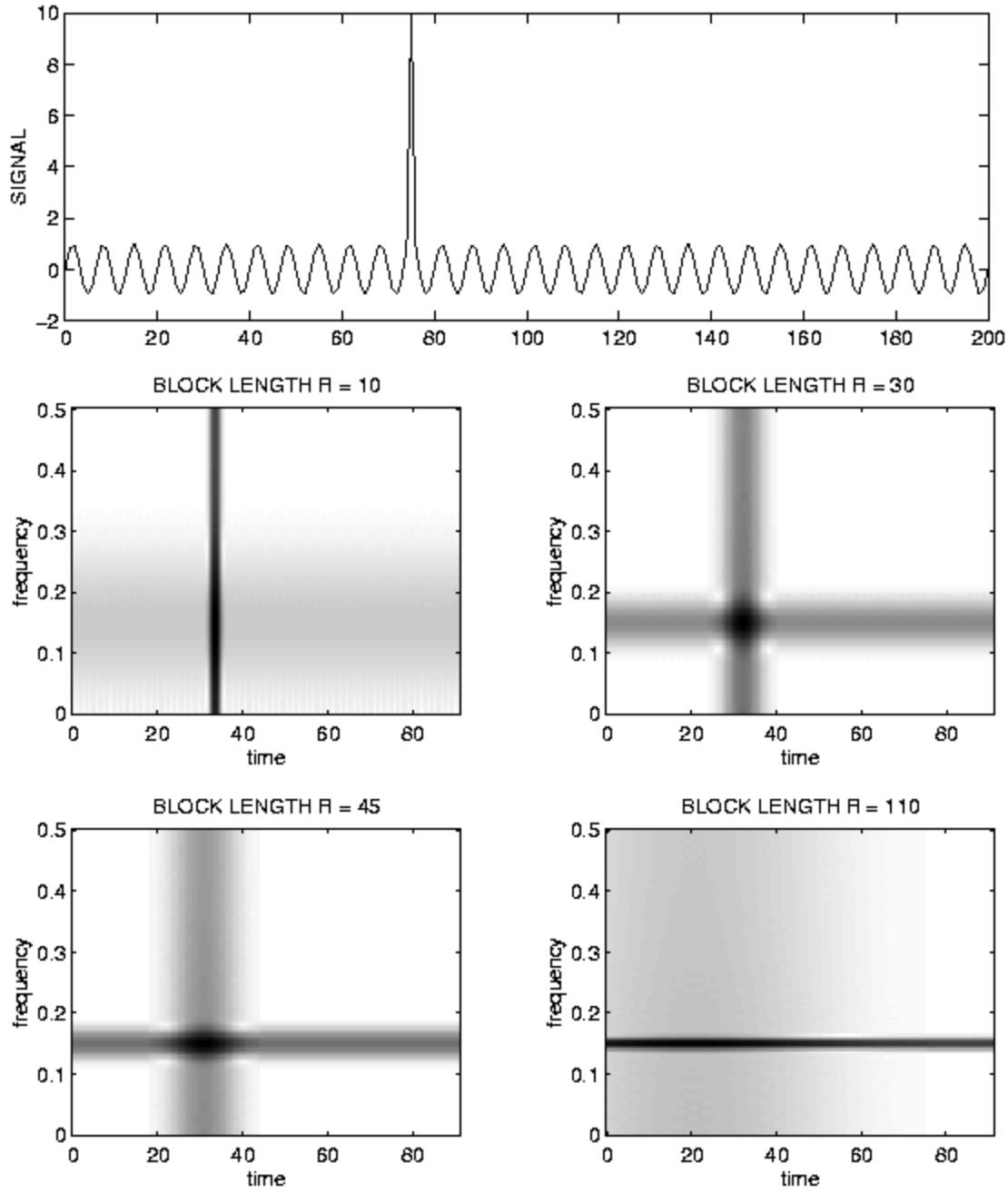


Wide-band spectrogram: better time resolution



Here is another example to illustrate the frequency/time resolution trade-off (See figures - [link], [link], and [link]).

Effect of Window Length R



Effect of L and N

A spectrogram is computed with different parameters:

$$L \in \{1, 10\}$$

$$N \in \{32, 256\}$$

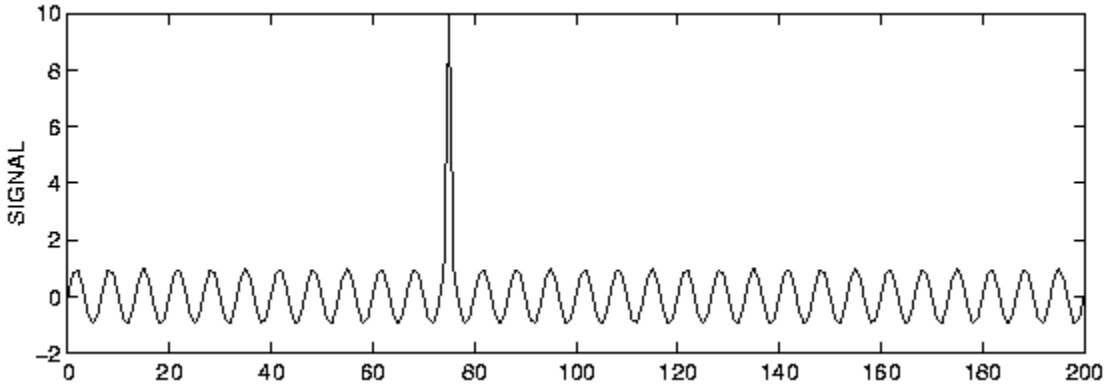
- L = time lapse between blocks.
- N = FFT length (Each block is zero-padded to length N.)

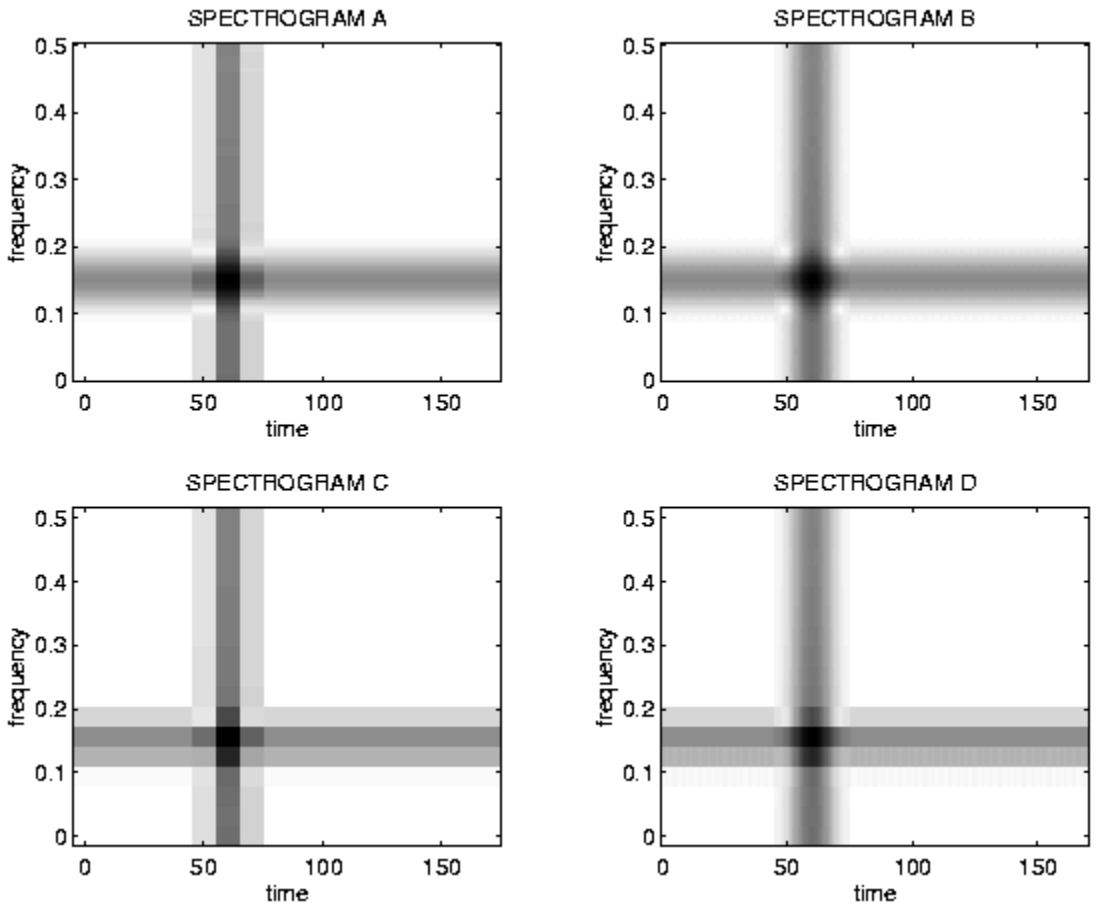
In each case, the block length is 30 samples.

Exercise:

Problem:

For each of the four spectrograms in [link] can you tell what L and N are?





Solution:

L and N do not effect the time resolution or the frequency resolution. They only affect the 'pixelation'.

Effect of R and L

Shown below are four spectrograms of the same signal. Each spectrogram is computed using a different set of parameters.

$$R \in \{120, 256, 1024\}$$

$$L \in \{35, 250\}$$

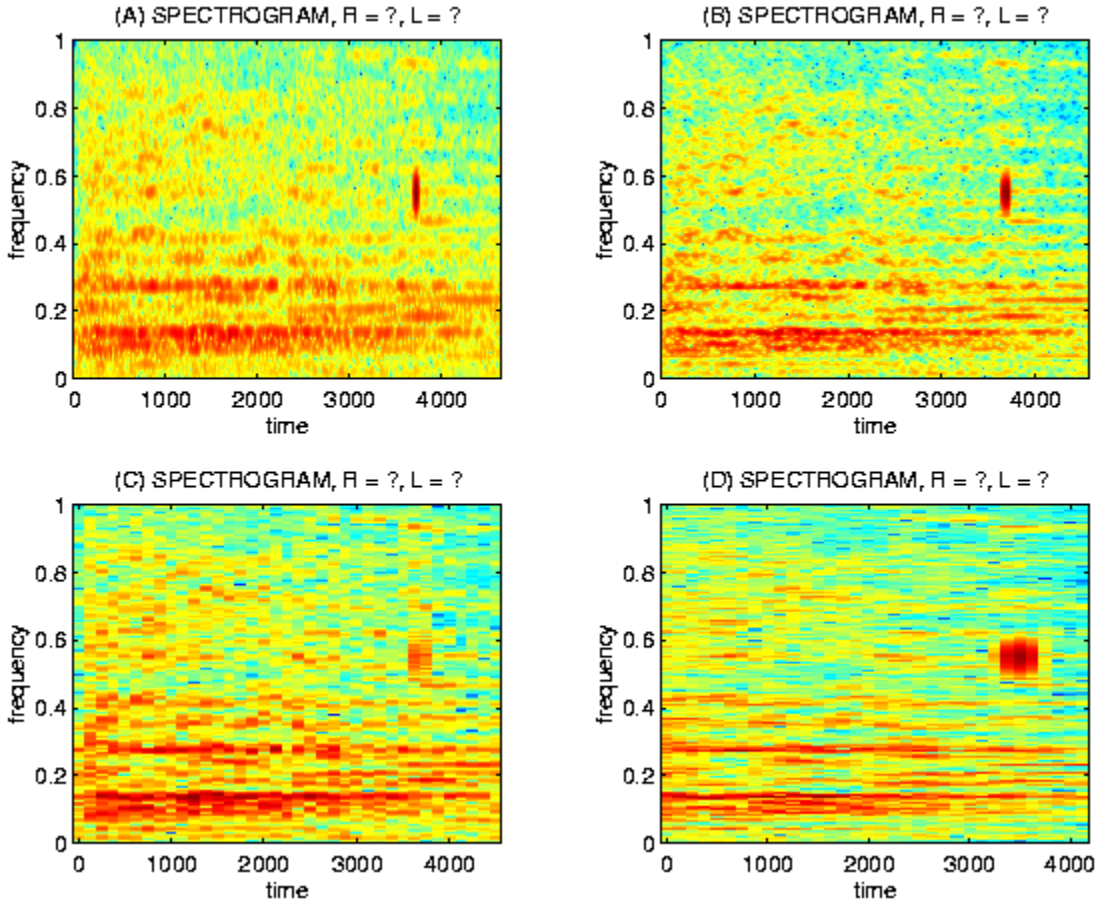
where

- R = block length
- L = time lapse between blocks.

Exercise:

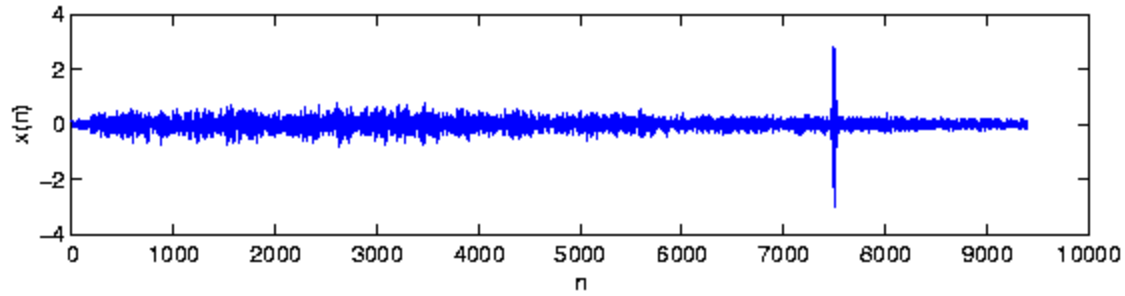
Problem:

For each of the four spectrograms in [link], match the above values of L and R.



Solution:

If you like, you may listen to this signal with the `soundsc` command; the data is in the file: `stft_data.m`. Here is a figure of the signal.


4
2
0
−2
−4
x(n)
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
n

Overview of Fast Fourier Transform (FFT) Algorithms

A fast Fourier transform, or FFT, is not a new transform, but is a computationally efficient algorithm for the computing the DFT. The length-N DFT, defined as

Equation:

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-\left(i \frac{2\pi nk}{N}\right)}$$

where $X(k)$ and $x(n)$ are in general complex-valued and $0 \leq k$, $n \leq N-1$, requires N complex multiplies to compute each $X(k)$. Direct computation of all N frequency samples thus requires N^2 complex multiplies and $N\left(N-1\right)$ complex additions. (This assumes precomputation of the DFT coefficients $W_N^{nk} \doteq e^{-\left(i \frac{2\pi nk}{N}\right)}$; otherwise, the cost is even higher.) For the large DFT lengths used in many applications, N^2 operations may be prohibitive. (For example, digital terrestrial television broadcast in Europe uses N = 2048 or 8192 OFDM channels, and the SETI project uses up to length-4194304 DFTs.) DFTs are thus almost always computed in practice by an FFT algorithm. FFTs are very widely used in signal processing, for applications such as spectrum analysis and digital filtering via fast convolution.

History of the FFT

It is now known that C.F. Gauss invented an FFT in 1805 or so to assist the computation of planetary orbits via discrete Fourier series. Various FFT algorithms were independently invented over the next two centuries, but FFTs achieved widespread awareness and impact only with the Cooley and Tukey algorithm published in 1965, which came at a time of increasing use of digital computers and when the vast range of applications of numerical Fourier techniques was becoming apparent. Cooley and Tukey's algorithm spawned a surge of research in FFTs and was also partly responsible for the emergence of Digital Signal Processing (DSP) as a distinct, recognized

discipline. Since then, many different algorithms have been rediscovered or developed, and efficient FFTs now exist for all DFT lengths.

Summary of FFT algorithms

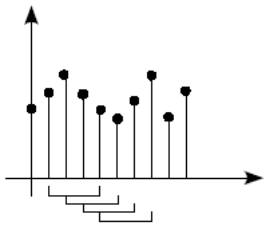
The main strategy behind most FFT algorithms is to factor a length-N DFT into a number of shorter-length DFTs, the outputs of which are reused multiple times (usually in additional short-length DFTs!) to compute the final results. The lengths of the short DFTs correspond to integer factors of the DFT length, N, leading to different algorithms for different lengths and factors. By far the most commonly used FFTs select $N = 2^M$ to be a power of two, leading to the very efficient power-of-two FFT algorithms, including the decimation-in-time radix-2 FFT and the decimation-in-frequency radix-2 FFT algorithms, the radix-4 FFT ($N = 4^M$), and the split-radix FFT. Power-of-two algorithms gain their high efficiency from extensive reuse of intermediate results and from the low complexity of length-2 and length-4 DFTs, which require no multiplications. Algorithms for lengths with repeated common factors (such as 2 or 4 in the radix-2 and radix-4 algorithms, respectively) require extra **twiddle factor** multiplications between the short-length DFTs, which together lead to a computational complexity of $O(N \log N)$, a very considerable savings over direct computation of the DFT.

The other major class of algorithms is the Prime-Factor Algorithms (PFA). In PFAs, the short-length DFTs must be of relatively prime lengths. These algorithms gain efficiency by reuse of intermediate computations and by eliminating twiddle-factor multiplies, but require more operations than the power-of-two algorithms to compute the short DFTs of various prime lengths. In the end, the computational costs of the prime-factor and the power-of-two algorithms are comparable for similar lengths, as illustrated in Choosing the Best FFT Algorithm. Prime-length DFTs cannot be factored into shorter DFTs, but in different ways both Rader's conversion and the chirp z-transform convert prime-length DFTs into convolutions of other lengths that can be computed efficiently using FFTs via fast convolution.

Some applications require only a few DFT frequency samples, in which case Goertzel's algorithm halves the number of computations relative to the DFT sum. Other applications involve successive DFTs of overlapped blocks of samples, for which the running FFT can be more efficient than separate FFTs of each block.

Running FFT

Some applications need DFT frequencies of the most recent N samples on an ongoing basis. One example is DTMF, or touch-tone telephone dialing, in which a detection circuit must constantly monitor the line for two simultaneous frequencies indicating that a telephone button is depressed. In such cases, most of the data in each successive block of samples is the same, and it is possible to efficiently update the DFT value from the previous sample to compute that of the current sample. [link] illustrates successive length-4 blocks of data for which successive DFT values may be needed. The **running FFT** algorithm described here can be used to compute successive DFT values at a cost of only two complex multiplies and additions per DFT frequency.



The running FFT efficiently computes DFT values for successive overlapped blocks of samples.

The running FFT algorithm is derived by expressing each DFT sample, $X_{n+1}(\omega_k)$, for the next block at time $n+1$ in terms of the previous value, $X_n(\omega_k)$, at time n.

$$X_n(\omega_k) = \sum_{p=0}^{N-1} x(n-p)e^{-(i\omega_k p)}$$

$$X_{n+1}(\omega_k) = \sum_{p=0}^{N-1} x(n+1-p)e^{-(i\omega_k p)}$$

Let $q = p - 1$:

$$X_{n+1}(\omega_k) = \sum_{q=-1}^{N-2} x(n-q)e^{-(i\omega_k(q-1))} = e^{i\omega_k}\sum_{q=0}^{N-2} x(n-q)e^{-(i\omega_k q)} + x(n+1)$$

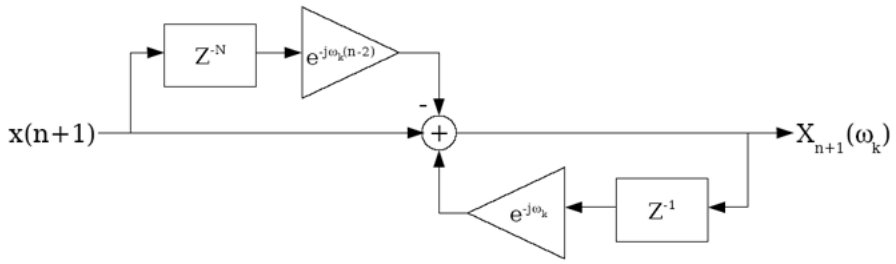
Now let's add and subtract $e^{-(i\omega_k(N-2))}x(n-N+1)$:
Equation:

$$\begin{aligned} X_{n+1}(\omega_k) &= e^{i\omega_k}\textstyle\sum_{q=0}^{N-2} x(n-q)e^{-(i\omega_k q)} + e^{i\omega_k}x(n-(N-1))e^{-(i\omega_k(N-1))} - e^{-(i\omega_k(N-2))}x(n-N+1) + \\ &= e^{i\omega_k}\textstyle\sum_{q=0}^{N-1} x(n-q)e^{-(i\omega_k)} + x(n+1) - e^{-(i\omega_k)}x(n-N+1) \\ &= e^{i\omega_k}X_n(\omega_k) + x(n+1) - e^{-(i\omega_k(N-2))}x(n-N+1) \end{aligned}$$

This running FFT algorithm requires only two complex multiplies and adds per update, rather than N if each DFT value were recomputed according to the DFT equation. Another advantage of this algorithm is that it works for

any ω_k, rather than just the standard DFT frequencies. This can make it advantageous for applications, such as DTMF detection, where only a few arbitrary frequencies are needed.

Successive computation of a specific DFT frequency for overlapped blocks can also be thought of as a length-N FIR filter. The running FFT is an efficient recursive implementation of this filter for this special case. [link] shows a block diagram of the running FFT algorithm. The running FFT is one way to compute DFT filterbanks. If a window other than rectangular is desired, a running FFT requires either a fast recursive implementation of the corresponding windowed, modulated impulse response, or it must have few non-zero coefficients so that it can be applied after the running FFT update via frequency-domain convolution. DFT-symmmetric raised-cosine windows are an example.



Block diagram of the running FFT computation, implemented as a recursive filter

Goertzel's Algorithm

Some applications require only a few DFT frequencies. One example is frequency-shift keying (FSK) demodulation, in which typically two frequencies are used to transmit binary data; another example is DTMF, or touch-tone telephone dialing, in which a detection circuit must constantly monitor the line for two simultaneous frequencies indicating that a telephone button is depressed. Goertzel's algorithm reduces the number of real-valued multiplications by almost a factor of two relative to direct computation via the DFT equation. Goertzel's algorithm is thus useful for computing a **few** frequency values; if many or most DFT values are needed, FFT algorithms that compute all DFT samples in $O(N \log N)$ operations are faster. Goertzel's algorithm can be derived by converting the DFT equation into an equivalent form as a convolution, which can be efficiently implemented as a digital filter. For increased clarity, in the equations below the complex exponential is denoted as $e^{-\left(i\frac{2\pi k}{N}\right)} = W_N^k$. Note that because W_N^{-Nk} always equals 1, the DFT equation can be rewritten as a convolution, or filtering operation:
Equation:

$$\begin{aligned} X(k) &= \textstyle\sum_{n=0}^{N-1} x(n) 1 W_N^{nk} \\ &= \textstyle\sum_{n=0}^{N-1} x(n) W_N^{-Nk} W_N^{nk} \\ &= \textstyle\sum_{n=0}^{N-1} x(n) W_N^{(N-n)(-k)} \\ &= \left(\left(\left(W_N^{-k} x(0) + x(1)\right) W_N^{-k} + x(2)\right) W_N^{-k} + \ldots + x(N-1)\right) W_N^{-k} \end{aligned}$$

Note that this last expression can be written in terms of a recursive difference equation

$$y(n) = W_N^{-k} y(n-1) + x(n)$$

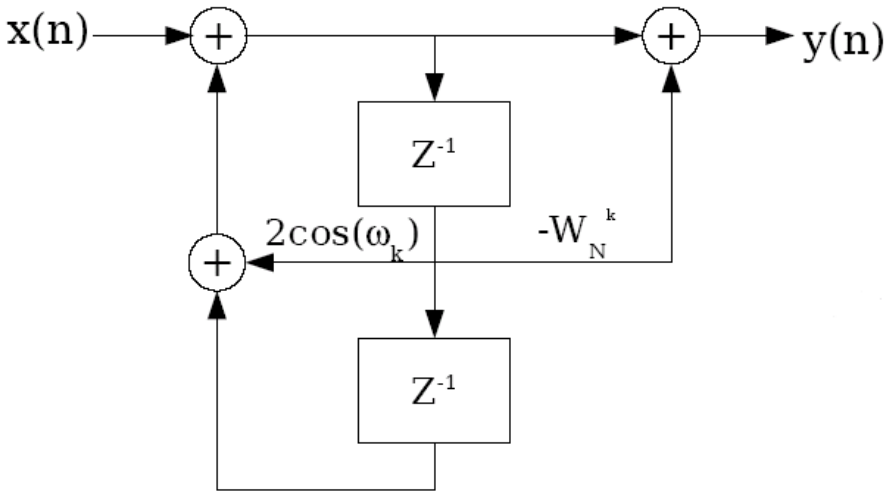
where $y(-1) = 0$. The DFT coefficient equals the output of the difference equation at time $n = N$:

$$X(k) = y(N)$$

Expressing the difference equation as a z-transform and multiplying both numerator and denominator by $1 - W_N^k z^{-1}$ gives the transfer function

$$\frac{Y(z)}{X(z)} = H(z) = \frac{1}{1 - W_N^{-k} z^{-1}} = \frac{1 - W_N^k z^{-1}}{1 - \left(\left(W_N^k + W_N^{-k}\right) z^{-1} - z^{-2}\right)} = \frac{1 - W_N^k z^{-1}}{1 - \left(2\cos\left(\frac{2\pi k}{N}\right) z^{-1} - z^{-2}\right)}$$

This system can be realized by the structure in [link]



We want $y(n)$ not for all n, but only for $n = N$. We can thus compute only the **recursive** part, or just the left side of the flow graph in [link], for $n = [0, 1, \ldots, N]$, which involves only a **real/complex** product rather than a complex/complex product as in a direct DFT, plus one complex multiply to get $y(N) = X(k)$.

Note: The input $x(N)$ at time $n = N$ must equal 0! A slightly more efficient alternate implementation that computes the full recursion only through $n = N - 1$ and combines the nonzero operations of the final recursion with the final complex multiply can be found here, complete with pseudocode (for real-valued data).

If the data are real-valued, only real/real multiplications and real additions are needed until the final multiply.

Note: The computational cost of Goertzel's algorithm is thus $2N + 2$ real multiplies and $4N - 2$ real adds, a reduction of almost a factor of two in the number of real multiplies relative to direct computation via the DFT equation. If the data are real-valued, this cost is almost halved again.

For certain frequencies, additional simplifications requiring even fewer multiplications are possible. (For example, for the DC ($k = 0$) frequency, all the multipliers equal 1 and only additions are needed.) A correspondence by C.G. Boncelet, Jr. describes some of these additional simplifications. Once again, Goertzel's and Boncelet's algorithms are efficient for a few DFT frequency samples; if more than $\log N$ frequencies are needed, $O(N \log N)$ FFT algorithms that compute all frequencies simultaneously will be more efficient.

Power-of-two FFTs

FFTs of length $N = 2^M$ equal to a power of two are, by far, the most commonly used. These algorithms are very efficient, relatively simple, and a single program can compute power-of-two FFTs of different lengths. As with most FFT algorithms, they gain their efficiency by computing **all** DFT points simultaneously through extensive reuse of intermediate computations; they are thus efficient when many DFT frequency samples are needed. The simplest power-of-two FFTs are the decimation-in-time radix-2 FFT and the decimation-in-frequency radix-2 FFT; they reduce the length-$N = 2^M$ DFT to a series of length-2 DFT computations with **twiddle-factor** complex multiplications between them. The radix-4 FFT algorithm similarly reduces a length-$N = 4^M$ DFT to a series of length-4 DFT computations with twiddle-factor multiplies in between. Radix-4 FFTs require only 75% as many complex multiplications as the radix-2 algorithms, although the number of complex additions remains the same. Radix-8 and higher-radix FFT algorithms can be derived using multi-dimensional index maps to reduce the computational complexity a bit more. However, the split-radix algorithm and its recent extensions combine the best elements of the radix-2 and radix-4 algorithms to obtain lower complexity than either or than any higher radix, requiring only two-thirds as many complex multiplies as the radix-2 algorithms. All of these algorithms obtain huge savings over direct computation of the DFT, reducing the complexity from $O\left(N^2\right)$ to $O(N \log N)$.

The efficiency of an FFT implementation depends on more than just the number of computations. Efficient FFT programming tricks can make up to a several-fold difference in the run-time of FFT programs. Alternate FFT structures can lead to a more convenient data flow for certain hardware. As discussed in choosing the best FFT algorithm, certain hardware is designed for, and thus most efficient for, FFTs of specific lengths or radices.

Decimation-in-time (DIT) Radix-2 FFT

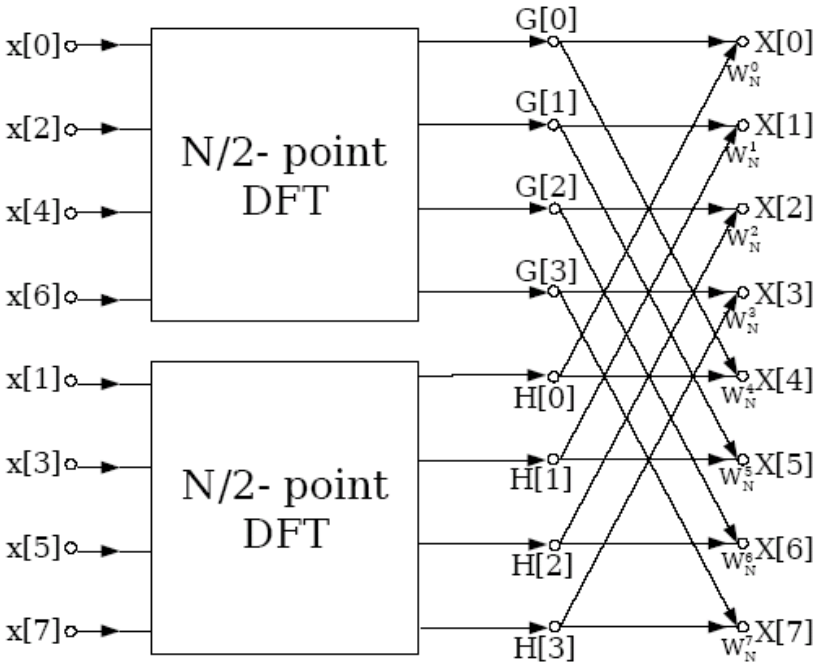
The radix-2 decimation-in-time and decimation-in-frequency fast Fourier transforms (FFTs) are the simplest FFT algorithms. Like all FFTs, they gain their speed by reusing the results of smaller, intermediate computations to compute multiple DFT frequency outputs.

Decimation in time

The radix-2 decimation-in-time algorithm rearranges the discrete Fourier transform (DFT) equation into two parts: a sum over the even-numbered discrete-time indices $n = [0, 2, 4, \ldots, N-2]$ and a sum over the odd-numbered indices $n = [1, 3, 5, \ldots, N-1]$ as in [link]:
Equation:

$$\begin{aligned} X(k) &= \sum_{n=0}^{N-1} x(n) e^{-\left(i\frac{2\pi nk}{N}\right)} \\ &= \sum_{n=0}^{\frac{N}{2}-1} x(2n) e^{-\left(i\frac{2\pi\times(2n)k}{N}\right)} + \sum_{n=0}^{\frac{N}{2}-1} x(2n+1) e^{-\left(i\frac{2\pi(2n+1)k}{N}\right)} \\ &= \sum_{n=0}^{\frac{N}{2}-1} x(2n) e^{-\left(i\frac{2\pi nk}{\frac{N}{2}}\right)} + e^{-\left(i\frac{2\pi k}{N}\right)} \sum_{n=0}^{\frac{N}{2}-1} x(2n+1) e^{-\left(i\frac{2\pi nk}{\frac{N}{2}}\right)} \\ &= \mathrm{DFT}_{\frac{N}{2}}\left[[x(0), x(2), \ldots, x(N-2)]\right] + W_N^k \, \mathrm{DFT}_{\frac{N}{2}}\left[[x(1), x(3), \ldots, x(N-1)]\right] \end{aligned}$$

The mathematical simplifications in [link] reveal that all DFT frequency outputs $X(k)$ can be computed as the sum of the outputs of two length-$\frac{N}{2}$ DFTs, of the even-indexed and odd-indexed discrete-time samples, respectively, where the odd-indexed short DFT is multiplied by a so-called **twiddle factor** term $W_N^k = e^{-\left(i\frac{2\pi k}{N}\right)}$. This is called a **decimation in time** because the time samples are rearranged in alternating groups, and a **radix-2** algorithm because there are two groups. [link] graphically illustrates this form of the DFT computation, where for convenience the frequency outputs of the length-$\frac{N}{2}$ DFT of the even-indexed time samples are denoted $G(k)$ and those of the odd-indexed samples as $H(k)$. Because of the periodicity with $\frac{N}{2}$ frequency samples of these length-$\frac{N}{2}$ DFTs, $G(k)$ and $H(k)$ can be used to compute **two** of the length-N DFT frequencies, namely $X(k)$ and $X\left(k+\frac{N}{2}\right)$, but with a different twiddle factor. This reuse of these short-length DFT outputs gives the FFT its computational savings.



Decimation in time of a length-N DFT into two length-$\frac{N}{2}$ DFTs followed by a combining stage.

Whereas direct computation of all N DFT frequencies according to the DFT equation would require N^2 complex multiplies and $N^2 - N$ complex additions (for complex-valued data), by reusing the results of the two short-length DFTs as illustrated in [link], the computational cost is now

New Operation Counts

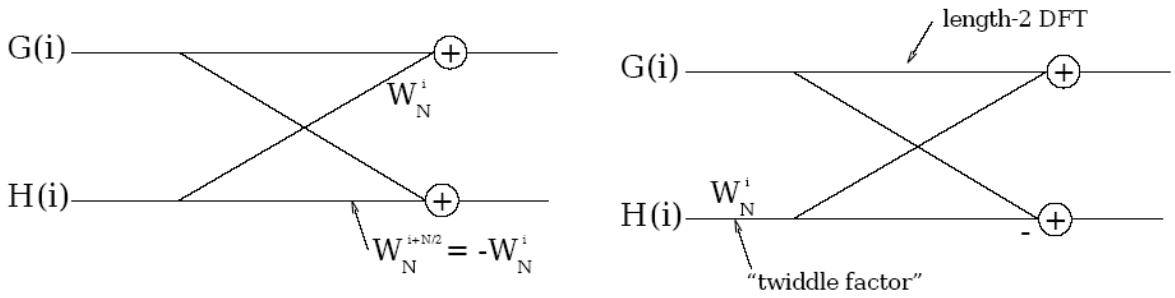
- $2\left(\frac{N}{2}\right)^2 + N = \frac{N^2}{2} + N$ complex multiplies
- $2\frac{N}{2}\left(\frac{N}{2} - 1\right) + N = \frac{N^2}{2}$ complex additions

This simple reorganization and reuse has reduced the total computation by almost a factor of two over direct DFT computation!

Additional Simplification

A basic **butterfly** operation is shown in [link], which requires only $\frac{N}{2}$ **twiddle-factor** multiplies per **stage**. It is worthwhile to note that, after merging the twiddle factors to a single term on the lower branch, the remaining butterfly is actually a length-2 DFT! The theory of multi-dimensional index maps shows that this must be the case, and that FFTs of any

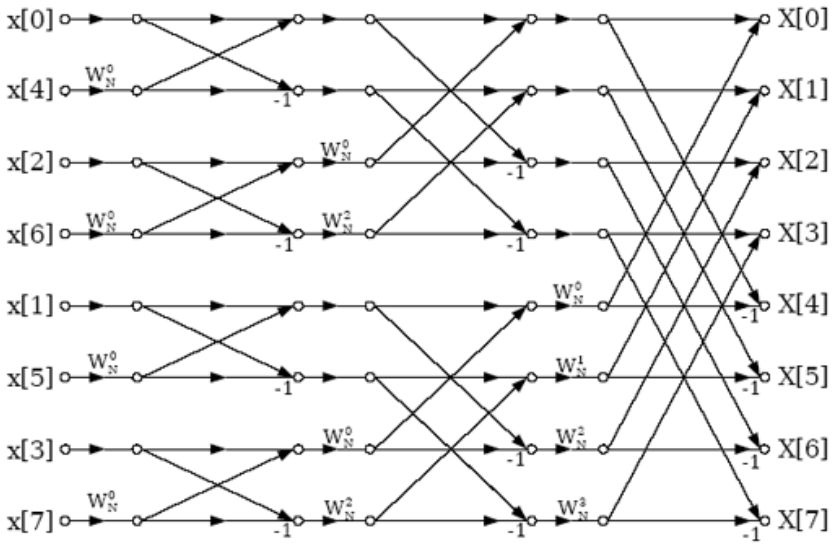
factorable length may consist of successive stages of shorter-length FFTs with twiddle-factor multiplications in between.



Radix-2 DIT butterfly simplification: both operations produce the same outputs

Radix-2 decimation-in-time FFT

The same radix-2 decimation in time can be applied recursively to the two length $\frac{N}{2}$ DFTs to save computation. When successively applied until the shorter and shorter DFTs reach length-2, the result is the radix-2 DIT FFT algorithm.



Radix-2 Decimation-in-Time FFT algorithm for a length-8 signal

The full radix-2 decimation-in-time decomposition illustrated in [link] using the simplified butterflies involves $M = \log_2 N$ stages, each with $\frac{N}{2}$ butterflies per stage. Each butterfly requires 1 complex multiply and 2 adds per butterfly. The total cost of the algorithm is thus
Computational cost of radix-2 DIT FFT

- $\frac{N}{2} \log_2 N$ complex multiplies
- $N \log_2 \mathrm{N}$ complex adds

This is a remarkable savings over direct computation of the DFT. For example, a length-1024 DFT would require 1048576 complex multiplications and 1047552 complex additions with direct computation, but only 5120 complex multiplications and 10240 complex additions using the radix-2 FFT, a savings by a factor of 100 or more. The relative savings increase with longer FFT lengths, and are less for shorter lengths.

Modest additional reductions in computation can be achieved by noting that certain twiddle factors, namely Using special butterflies for W_N^0, $W_N^{\frac{N}{2}}$, $W_N^{\frac{N}{4}}$, $W_N^{\frac{N}{8}}$, $W_N^{\frac{3N}{8}}$, require no multiplications, or fewer real multiplies than other ones. By implementing special butterflies for these twiddle factors as discussed in FFT algorithm and programming tricks, the computational cost of the radix-2 decimation-in-time FFT can be reduced to

- $2N \log_2 N - 7N + 12$ real multiplies
- $3N \log_2 N - 3N + 4$ real additions

Note: In a decimation-in-time radix-2 FFT as illustrated in [link], the input is in **bit-reversed** order (hence "decimation-in-time"). That is, if the time-sample index n is written as a binary number, the order is that binary number reversed. The bit-reversal process is illustrated for a length-$N = 8$ example below.

Example:
N=8

In-order index	In-order index in binary	Bit-reversed binary	Bit-reversed index

In-order index	In-order index in binary	Bit-reversed binary	Bit-reversed index
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

It is important to note that, if the input signal data are placed in bit-reversed order before beginning the FFT computations, the outputs of each butterfly throughout the computation can be placed in the same memory locations from which the inputs were fetched, resulting in an **in-place algorithm** that requires no extra memory to perform the FFT. Most FFT implementations are in-place, and overwrite the input data with the intermediate values and finally the output.

Example FFT Code

The following function, written in the C programming language, implements a radix-2 decimation-in-time FFT. It is designed for computing the DFT of complex-valued inputs to produce complex-valued outputs, with the real and imaginary parts of each number stored in separate double-precision floating-point arrays. It is an in-place algorithm, so the intermediate and final output values are stored in the same array as the input data, which is overwritten. After initializations, the program first bit-reverses the discrete-time samples, as is typical with a decimation-in-time algorithm (but see alternate FFT structures for DIT algorithms with other input orders), then computes the FFT in stages according to the above description.

Ihis FFT program uses a standard three-loop structure for the main FFT computation. The outer loop steps through the stages (each column in [link]); the middle loop steps through "**flights**" (butterflies with the same twiddle factor from each short-length DFT at each stage), and the inner loop steps through the individual butterflies. This ordering minimizes the number of fetches or computations of the twiddle-factor values. Since the bit-reverse of a bit-

reversed index is the original index, bit-reversal can be performed fairly simply by swapping pairs of data.

Note: While of $O(N \log N)$ complexity and thus much faster than a direct DFT, this simple program is optimized for clarity, not for speed. A speed-optimized program making use of additional efficient FFT algorithm and programming tricks will compute a DFT several times faster on most machines.

```
/*************************************************************/
/* fft.c                                                     */
/* (c) Douglas L. Jones                                      */
/* University of Illinois at Urbana-Champaign                */
/* January 19, 1992                                          */
/*                                                           */
/*   fft: in-place radix-2 DIT DFT of a complex input        */
/*                                                           */
/*   input:                                                  */
/* n: length of FFT: must be a power of two                  */
/* m: n = 2**m                                               */
/*   input/output                                            */
/* x: double array of length n with real part of data        */
/* y: double array of length n with imag part of data        */
/*                                                           */
/*   Permission to copy and use this program is granted      */
/*   under a Creative Commons "Attribution" license          */
/*   http://creativecommons.org/licenses/by/1.0/             */
/*************************************************************/
fft(n,m,x,y)
int n,m;
double x[],y[];
{
int i,j,k,n1,n2;
double c,s,e,a,t1,t2;


j = 0; /* bit-reverse */
n2 = n/2;
for (i=1; i < n - 1; i++)
{
  n1 = n2;
  while ( j >= n1 )
```

```
   {
    j = j - n1;
    n1 = n1/2;
   }
  j = j + n1;

  if (i < j)
   {
    t1 = x[i];
    x[i] = x[j];
    x[j] = t1;
    t1 = y[i];
    y[i] = y[j];
    y[j] = t1;
   }
}


n1 = 0; /* FFT */
n2 = 1;

for (i=0; i < m; i++)
{
  n1 = n2;
  n2 = n2 + n2;
  e = -6.283185307179586/n2;
  a = 0.0;

  for (j=0; j < n1; j++)
   {
    c = cos(a);
    s = sin(a);
    a = a + e;

    for (k=j; k < n; k=k+n2)
     {
      t1 = c*x[k+n1] - s*y[k+n1];
      t2 = s*x[k+n1] + c*y[k+n1];
      x[k+n1] = x[k] - t1;
      y[k+n1] = y[k] - t2;
      x[k] = x[k] + t1;
      y[k] = y[k] + t2;
     }
   }
}
```

```
return;
}
```

Decimation-in-Frequency (DIF) Radix-2 FFT

The radix-2 decimation-in-frequency and decimation-in-time fast Fourier transforms (FFTs) are the simplest FFT algorithms. Like all FFTs, they compute the discrete Fourier transform (DFT)
Equation:

$$\begin{aligned} X(k) &= \sum_{n=0}^{N-1} x(n) e^{-\left(i \frac{2\pi n k}{N}\right)} \\ &= \sum_{n=0}^{N-1} x(n) W_N^{nk} \end{aligned}$$

where for notational convenience $W_N^k = e^{-\left(i \frac{2\pi k}{N}\right)}$. FFT algorithms gain their speed by reusing the results of smaller, intermediate computations to compute multiple DFT frequency outputs.

Decimation in frequency

The radix-2 decimation-in-frequency algorithm rearranges the discrete Fourier transform (DFT) equation into two parts: computation of the even-numbered discrete-frequency indices $X(k)$ for $k = [0, 2, 4, \ldots, N-2]$ (or $X(2r)$ as in [link]) and computation of the odd-numbered indices $k = [1, 3, 5, \ldots, N-1]$ (or $X(2r+1)$ as in [link])
Equation:

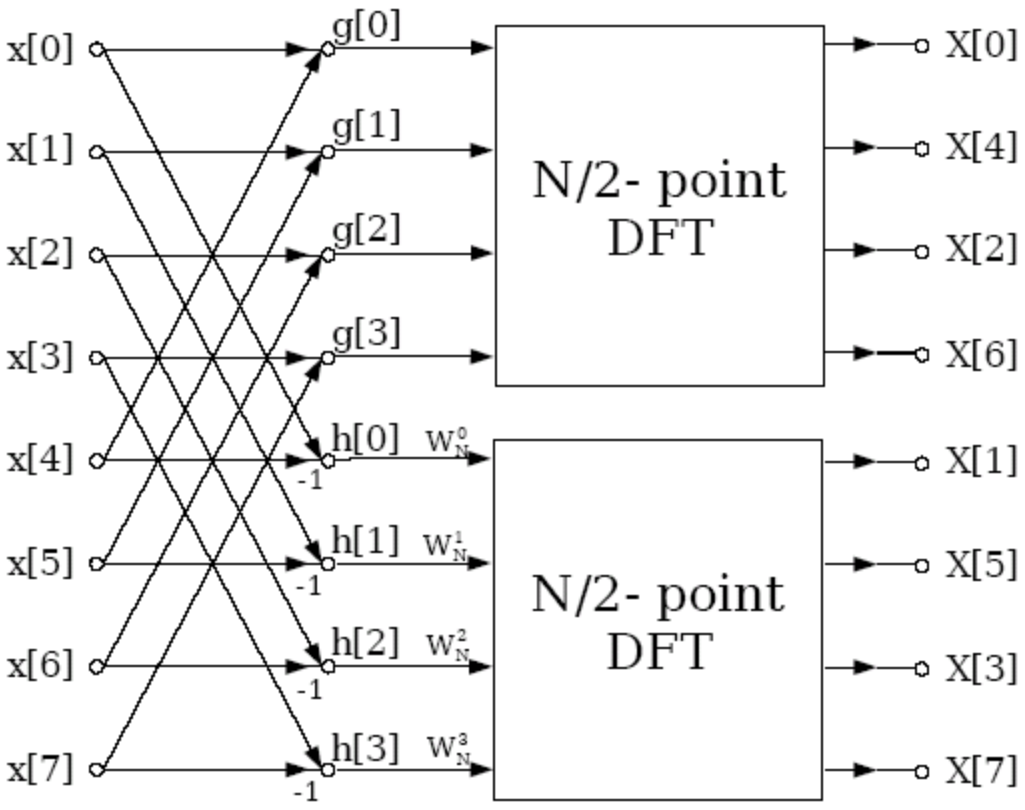
$$\begin{aligned} X(2r) &= \sum_{n=0}^{N-1} x(n) W_N^{2rn} \\ &= \sum_{n=0}^{\frac{N}{2}-1} x(n) W_N^{2rn} + \sum_{n=0}^{\frac{N}{2}-1} x\left(n + \tfrac{N}{2}\right) W_N^{2r\left(n+\frac{N}{2}\right)} \\ &= \sum_{n=0}^{\frac{N}{2}-1} x(n) W_N^{2rn} + \sum_{n=0}^{\frac{N}{2}-1} x\left(n + \tfrac{N}{2}\right) W_N^{2rn} 1 \\ &= \sum_{n=0}^{\frac{N}{2}-1} \left(x(n) + x\left(n + \tfrac{N}{2}\right)\right) W_{\frac{N}{2}}^{rn} \\ &= \mathrm{DFT}_{\frac{N}{2}} \left[x(n) + x\left(n + \tfrac{N}{2}\right)\right] \end{aligned}$$

Equation:

$$
\begin{aligned}
X(2r+1) &= \textstyle\sum_{n=0}^{N-1} x(n) W_N^{(2r+1)n} \\
&= \textstyle\sum_{n=0}^{\frac{N}{2}-1} \left(x(n) + W_N^{\frac{N}{2}} x\left(n+\frac{N}{2}\right) \right) W_N^{(2r+1)n} \\
&= \textstyle\sum_{n=0}^{\frac{N}{2}-1} \left(\left(x(n) - x\left(n+\frac{N}{2}\right) \right) W_N^n \right) W_{\frac{N}{2}}^{rn} \\
&= \mathrm{DFT}_{\frac{N}{2}} \left[\left(x(n) - x\left(n+\frac{N}{2}\right) \right) W_N^n \right]
\end{aligned}
$$

The mathematical simplifications in [link] and [link] reveal that both the even-indexed and odd-indexed frequency outputs $X(k)$ can each be computed by a length-$\frac{N}{2}$ DFT. The inputs to these DFTs are sums or differences of the first and second halves of the input signal, respectively, where the input to the short DFT producing the odd-indexed frequencies is multiplied by a so-called **twiddle factor** term $W_N^k = e^{-\left(i\frac{2\pi k}{N}\right)}$. This is called a **decimation in frequency** because the frequency samples are computed separately in alternating groups, and a **radix-2** algorithm because there are two groups. [link] graphically illustrates this form of the DFT computation. This conversion of the full DFT into a series of shorter DFTs with a simple preprocessing step gives the decimation-in-frequency FFT its computational savings.

Decimation in frequency of a length-N DFT into two length-$\frac{N}{2}$ DFTs preceded by a preprocessing stage.



Whereas direct computation of all N DFT frequencies according to the DFT equation would require N^2 complex multiplies and $N^2 - N$ complex additions (for complex-valued data), by breaking the computation into two short-length DFTs with some preliminary combining of the data, as illustrated in [link], the computational cost is now

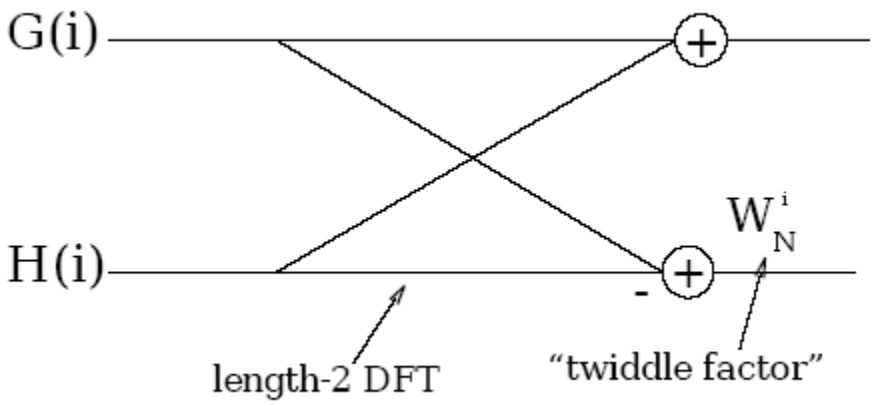
New Operation Counts

- $2\left(\frac{N}{2}\right)^2 + N = \frac{N^2}{2} + \frac{N}{2}$ complex multiplies
- $2\frac{N}{2}\left(\frac{N}{2} - 1\right) + N = \frac{N^2}{2}$ complex additions

This simple manipulation has reduced the total computational cost of the DFT by almost a factor of two!

The initial combining operations for both short-length DFTs involve parallel groups of two time samples, $x(n)$ and $x\left(n + \frac{N}{2}\right)$. One of these so-called **butterfly** operations is illustrated in [link]. There are $\frac{N}{2}$ butterflies

per **stage**, each requiring a complex addition and subtraction followed by one **twiddle-factor** multiplication by $W_N^n = e^{-\left(i\frac{2\pi n}{N}\right)}$ on the lower output branch.

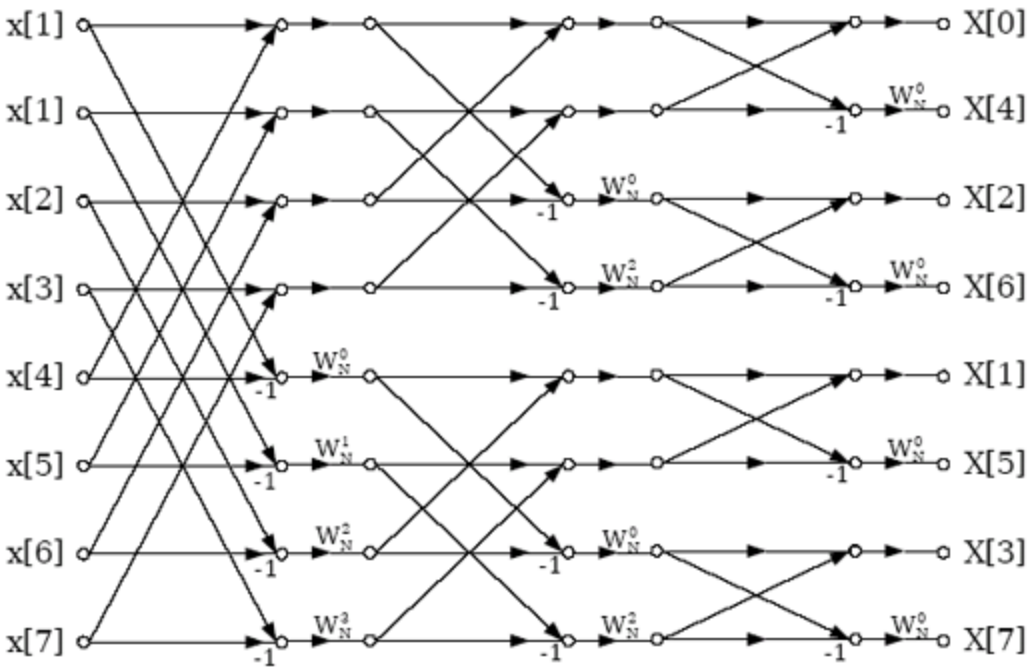


DIF butterfly: twiddle factor after length-2 DFT

It is worthwhile to note that the initial add/subtract part of the DIF butterfly is actually a length-2 DFT! The theory of multi-dimensional index maps shows that this must be the case, and that FFTs of any factorable length may consist of successive stages of shorter-length FFTs with twiddle-factor multiplications in between. It is also worth noting that this butterfly differs from the decimation-in-time radix-2 butterfly in that the twiddle factor multiplication occurs **after** the combining.

Radix-2 decimation-in-frequency algorithm

The same radix-2 decimation in frequency can be applied recursively to the two length-$\frac{N}{2}$ DFTs to save additional computation. When successively applied until the shorter and shorter DFTs reach length-2, the result is the radix-2 decimation-in-frequency FFT algorithm.



Radix-2 decimation-in-frequency FFT for a length-8 signal

The full radix-2 decimation-in-frequency decomposition illustrated in [link] requires $M = \log_2 N$ stages, each with $\frac{N}{2}$ butterflies per stage. Each butterfly requires 1 complex multiply and 2 adds per butterfly. The total cost of the algorithm is thus

Computational cost of radix-2 DIF FFT

- $\frac{N}{2} \log_2 N$ complex multiplies
- $N \log_2 \mathrm{N}$ complex adds

This is a remarkable savings over direct computation of the DFT. For example, a length-1024 DFT would require 1048576 complex multiplications and 1047552 complex additions with direct computation, but only 5120 complex multiplications and 10240 complex additions using the radix-2 FFT, a savings by a factor of 100 or more. The relative savings increase with longer FFT lengths, and are less for shorter lengths. Modest

additional reductions in computation can be achieved by noting that certain twiddle factors, namely W_N^0, $W_N^{\frac{N}{2}}$, $W_N^{\frac{N}{4}}$, $W_N^{\frac{N}{8}}$, $W_N^{\frac{3N}{8}}$, require no multiplications, or fewer real multiplies than other ones. By implementing special butterflies for these twiddle factors as discussed in FFT algorithm and programming tricks, the computational cost of the radix-2 decimation-in-frequency FFT can be reduced to

- $2N \log_2 N - 7N + 12$ real multiplies
- $3N \log_2 N - 3N + 4$ real additions

The decimation-in-frequency FFT is a flow-graph reversal of the decimation-in-time FFT: it has the same twiddle factors (in reverse pattern) and the same operation counts.

Note: In a decimation-in-frequency radix-2 FFT as illustrated in [link], the output is in **bit-reversed** order (hence "decimation-in-frequency"). That is, if the frequency-sample index n is written as a binary number, the order is that binary number reversed. The bit-reversal process is illustrated here.

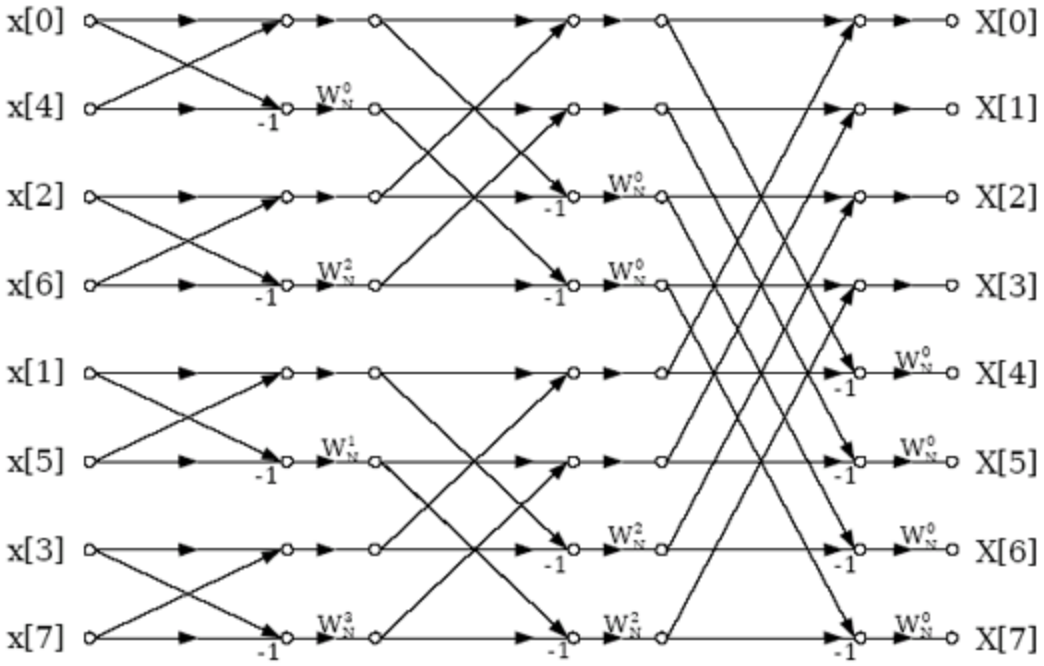
It is important to note that, if the input data are in order before beginning the FFT computations, the outputs of each butterfly throughout the computation can be placed in the same memory locations from which the inputs were fetched, resulting in an **in-place algorithm** that requires no extra memory to perform the FFT. Most FFT implementations are in-place, and overwrite the input data with the intermediate values and finally the output.

Alternate FFT Structures

Bit-reversing the input in decimation-in-time (DIT) FFTs or the output in decimation-in-frequency (DIF) FFTs can sometimes be inconvenient or inefficient. For such situations, alternate FFT structures have been developed. Such structures involve the same mathematical computations as the standard algorithms, but alter the memory locations in which intermediate values are stored or the order of computation of the FFT butterflies.

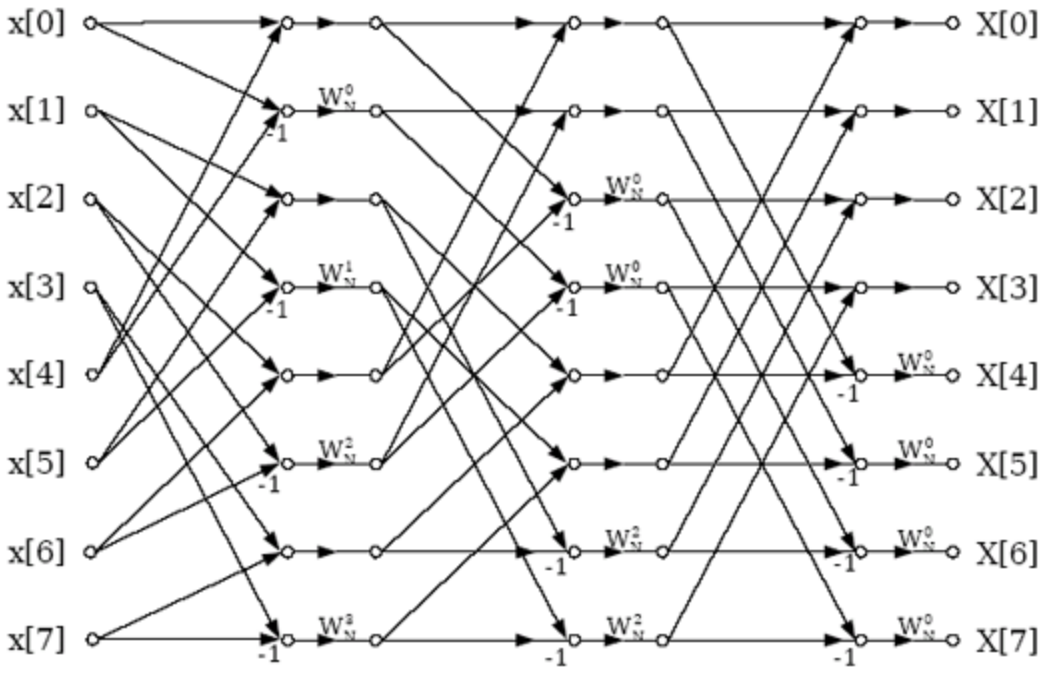
The structure in [link] computes a decimation-in-frequency FFT, but remaps the memory usage so that the **input** is bit-reversed, and the output is in-order as in the conventional decimation-in-time FFT. This alternate structure is still considered a DIF FFT because the twiddle factors are applied as in the DIF FFT. This structure is useful if for some reason the DIF butterfly is preferred but it is easier to bit-reverse the input.

Decimation-in-frequency radix-2 FFT with bit-reversed **input**. This is an in-place algorithm in which the same memory can be reused throughout the computation.



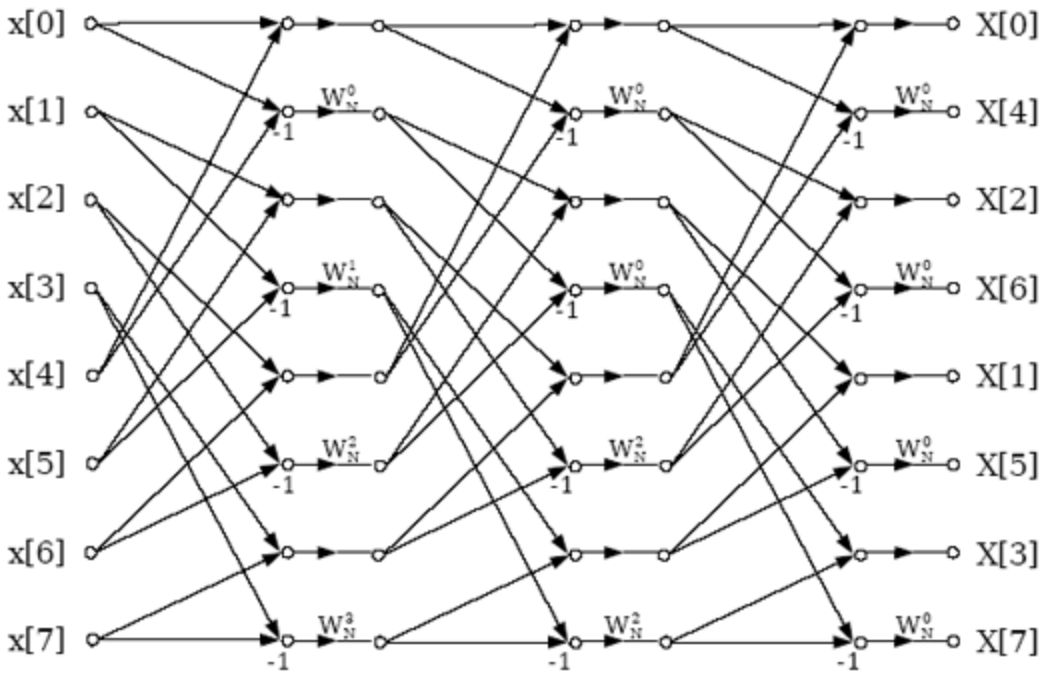
There is a similar structure for the decimation-in-time FFT with in-order inputs and bit-reversed frequencies. This structure can be useful for fast convolution on machines that favor decimation-in-time algorithms because the filter can be stored in bit-reverse order, and then the inverse FFT returns an in-order result without ever bit-reversing any data. As discussed in Efficient FFT Programming Tricks, this may save several percent of the execution time.

The structure in [link] implements a decimation-in-frequency FFT that has both input and output in order. It thus avoids the need for bit-reversing altogether. Unfortunately, it destroys the in-place structure somewhat, making an FFT program more complicated and requiring more memory; on most machines the resulting cost exceeds the benefits. This structure can be computed in place if **two** butterflies are computed simultaneously.



Decimation-in-frequency radix-2 FFT with in-order input and output. It can be computed in-place if two butterflies are computed simultaneously.

The structure in [link] has a constant geometry; the connections between memory locations are identical in each FFT stage. Since it is not in-place and requires bit-reversal, it is inconvenient for software implementation, but can be attractive for a highly parallel hardware implementation because the connections between stages can be hardwired. An analogous structure exists that has bit-reversed inputs and in-order outputs.



This constant-geometry structure has the same interconnect pattern from stage to stage. This structure is sometimes useful for special hardware.

Radix-4 FFT Algorithms

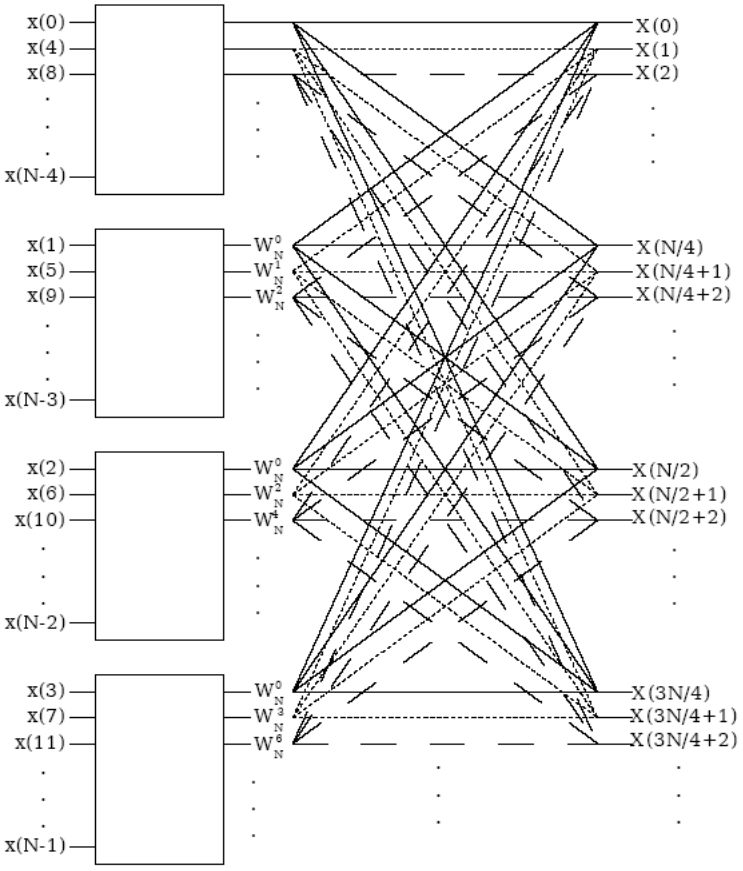
The radix-4 decimation-in-time and decimation-in-frequency fast Fourier transforms (FFTs) gain their speed by reusing the results of smaller, intermediate computations to compute multiple DFT frequency outputs. The radix-4 decimation-in-time algorithm rearranges the discrete Fourier transform (DFT) equation into four parts: sums over all groups of every fourth discrete-time index $n = [0, 4, 8, \ldots, N-4]$, $n = [1, 5, 9, \ldots, N-3]$, $n = [2, 6, 10, \ldots, N-2]$ and $n = [3, 7, 11, \ldots, N-1]$ as in [link]. (This works out only when the FFT length is a multiple of four.) Just as in the radix-2 decimation-in-time FFT, further mathematical manipulation shows that the length-N DFT can be computed as the sum of the outputs of four length-$\frac{N}{4}$ DFTs, of the even-indexed and odd-indexed discrete-time samples, respectively, where three of them are multiplied by so-called **twiddle factors** $W_N^k = e^{-\left(i\frac{2\pi k}{N}\right)}$, W_N^{2k}, and W_N^{3k}.

Equation:

$$\begin{aligned} X(k) &= \sum_{\mathrm{n}=0}^{N-1} x(n) e^{-\left(i\frac{2\pi nk}{N}\right)} \\ &= \sum_{n=0}^{\frac{N}{4}-1} x(4n) e^{-\left(i\frac{2\pi\times(4n)k}{N}\right)} + \sum_{n=0}^{\frac{N}{4}-1} x(4n+1) e^{-\left(i\frac{2\pi(4n+1)k}{N}\right)} + \sum_{n=0}^{\frac{N}{4}-1} x(4n+2) e^{-\left(i\frac{2\pi(4n+2)k}{N}\right)} + \sum_{n=0}^{\frac{N}{4}-} \\ &= \mathrm{DFT}_{\frac{N}{4}}[x(4n)] + W_N^k \mathrm{DFT}_{\frac{N}{4}}[x(4n+1)] + W_N^{2k} \mathrm{DFT}_{\frac{N}{4}}[x(4n+2)] + W_N^{3k} \mathrm{DFT}_{\frac{N}{4}}[x(4n+3)] \end{aligned}$$

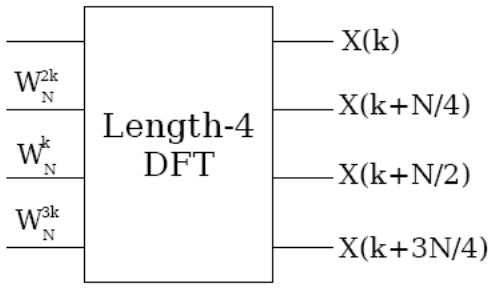
This is called a **decimation in time** because the time samples are rearranged in alternating groups, and a **radix-4** algorithm because there are four groups. [link] graphically illustrates this form of the DFT computation.

Radix-4 DIT structure



Decimation in time of a length-N DFT into four length-$\frac{N}{4}$ DFTs followed by a combining stage.

Due to the periodicity with $\frac{N}{4}$ of the short-length DFTs, their outputs for frequency-sample k are reused to compute $X(k)$, $X\left(k+\frac{N}{4}\right)$, $X\left(k+\frac{N}{2}\right)$, and $X\left(k+\frac{3N}{4}\right)$. It is this reuse that gives the radix-4 FFT its efficiency. The computations involved with each group of four frequency samples constitute the **radix-4 butterfly**, which is shown in [link]. Through further rearrangement, it can be shown that this computation can be simplified to three twiddle-factor multiplies and a length-4 DFT! The theory of multi-dimensional index maps shows that this must be the case, and that FFTs of any factorable length may consist of successive stages of shorter-length FFTs with twiddle-factor multiplications in between. The length-4 DFT requires no multiplies and only eight complex additions (this efficient computation can be derived using a radix-2 FFT).



[missing_resource: imageX.png]

The radix-4 DIT butterfly can be simplified to a length-4 DFT preceded by three twiddle-factor multiplies.

If the FFT length $N = 4^M$, the shorter-length DFTs can be further decomposed recursively in the same manner to produce the full **radix-4 decimation-in-time FFT**. As in the radix-2 decimation-in-time FFT, each stage of decomposition creates additional savings in computation. To determine the total computational cost of the radix-4 FFT, note that there are $M = \log_4 N = \frac{\log_2 N}{2}$ stages, each with $\frac{N}{4}$ butterflies per stage. Each radix-4 butterfly requires 3 complex multiplies and 8 complex additions. The total cost is then

Radix-4 FFT Operation Counts

- $3\frac{N}{4}\frac{\log_2 N}{2} = \frac{3}{8}N\log_2 N$ complex multiplies (75% of a radix-2 FFT)
- $8\frac{N}{4}\frac{\log_2 N}{2} = N\log_2 N$ complex adds (same as a radix-2 FFT)

The radix-4 FFT requires only 75% as many complex multiplies as the radix-2 FFTs, although it uses the same number of complex additions. These additional savings make it a widely-used FFT algorithm.

The decimation-in-time operation regroups the input samples at each successive stage of decomposition, resulting in a "digit-reversed" input order. That is, if the time-sample index n is written as a base-4 number, the order is that base-4 number reversed. The digit-reversal process is illustrated for a length-$N = 64$ example below.

Example:
N = 64 = 4^3

Original Number	Original Digit Order	Reversed Digit Order	Digit-Reversed Number
0	000	000	0
1	001	100	16
2	002	200	32
3	003	300	48
4	010	010	4
5	011	110	20
⋮	⋮	⋮	⋮

It is important to note that, if the input signal data are placed in digit-reversed order before beginning the FFT computations, the outputs of each butterfly throughout the computation can be placed in the same memory locations from which the inputs were fetched, resulting in an **in-place algorithm** that requires no extra memory to perform the FFT. Most FFT implementations are in-place, and overwrite the input data with the intermediate values and finally the output. A slight rearrangement within the radix-4 FFT introduced by Burrus allows the inputs to be arranged in bit-reversed rather than digit-reversed order.

A radix-4 decimation-in-frequency FFT can be derived similarly to the radix-2 DIF FFT, by separately computing all four groups of every fourth **output** frequency sample. The DIF radix-4 FFT is a flow-graph reversal of the DIT radix-4 FFT, with the same operation counts and twiddle factors in the reversed order. The output ends up in digit-reversed order for an in-place DIF algorithm.

Exercise:

> **Problem:** How do we derive a radix-4 algorithm when $N = 4^M 2$?

> **Solution:**
>
> Perform a radix-2 decomposition for one stage, then radix-4 decompositions of all subsequent shorter-length DFTs.

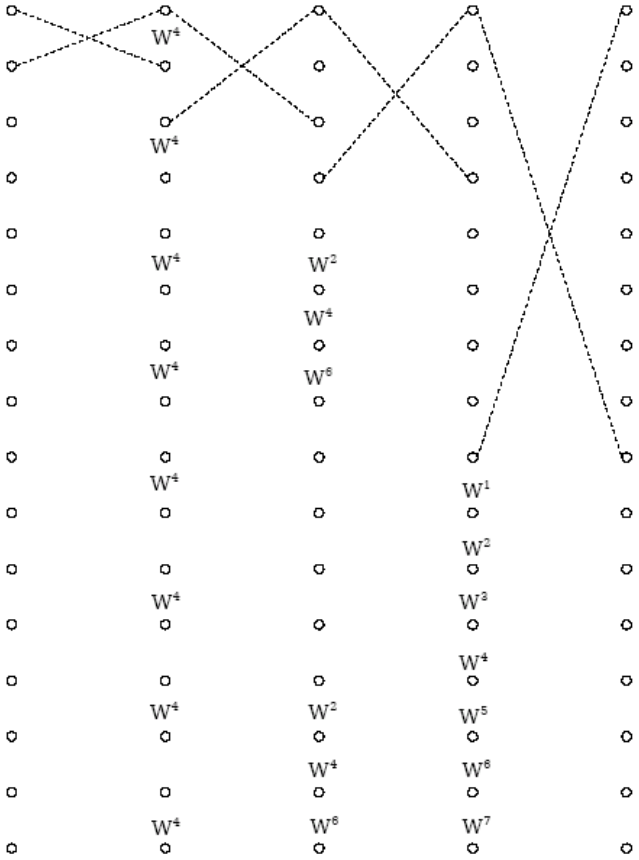
Split-radix FFT Algorithms

The split-radix algorithm, first clearly described and named by Duhamel and Hollman in 1984, required fewer total multiply and add operations operations than any previous power-of-two algorithm. (Yavne first derived essentially the same algorithm in 1968, but the description was so atypical that the work was largely neglected.) For a time many FFT experts thought it to be optimal in terms of total complexity, but even more efficient variations have more recently been discovered by Johnson and Frigo.

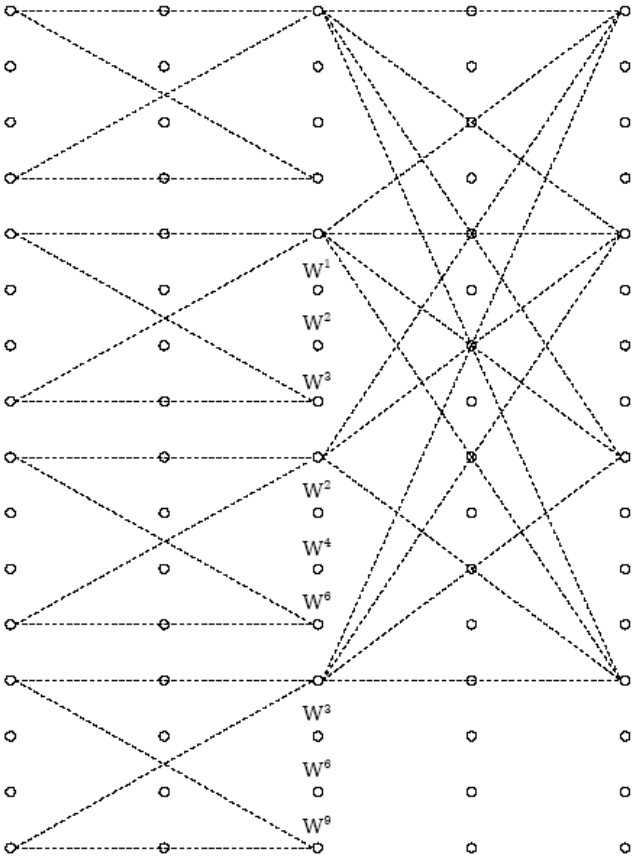
The split-radix algorithm can be derived by careful examination of the radix-2 and radix-4 flowgraphs as in Figure 1 below. While in most places the radix-4 algorithm has fewer nontrivial twiddle factors, in some places the radix-2 actually lacks twiddle factors present in the radix-4 structure or those twiddle factors simplify to multiplication by $-i$, which actually requires only additions. By mixing radix-2 and radix-4 computations appropriately, an algorithm of lower complexity than either can be derived.

Motivation for split-radix algorithm

radix-2

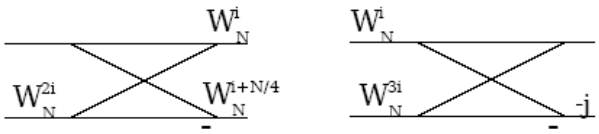


radix-4



See Decimation-in-Time (DIT) Radix-2 FFT and Radix-4 FFT Algorithms for more information on these algorithms.

An alternative derivation notes that radix-2 butterflies of the form shown in Figure 2 can merge twiddle factors from two successive stages to eliminate one-third of them; hence, the split-radix algorithm requires only about two-thirds as many multiplications as a radix-2 FFT.



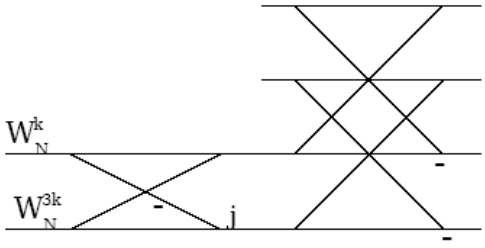
Note that these two butterflies are equivalent

The split-radix algorithm can also be derived by mixing the radix-2 and radix-4 decompositions.
Equation:

DIT Split-radix derivation

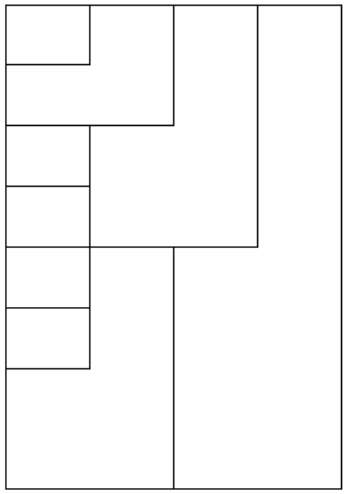
$$\begin{aligned} X(k) &= \sum_{n=0}^{\frac{N}{2}-1} x(2n) e^{-\left(i\frac{2\pi\times(2n)k}{N}\right)} + \sum_{n=0}^{\frac{N}{4}-1} x(4n+1) e^{-\left(i\frac{2\pi(4n+1)k}{N}\right)} + \sum_{n=0}^{\frac{N}{4}-1} x(4n+3) e^{-\left(i\frac{2\pi(4n+3)k}{N}\right)} \\ &= \mathrm{DFT}_{\frac{N}{2}}\left[x(2n)\right] + W_N^k \mathrm{DFT}_{\frac{N}{4}}(x(4n+1)) + W_N^{3k} \mathrm{DFT}_{\frac{N}{4}}(x(4n+3)) \end{aligned}$$

Figure 3 illustrates the resulting split-radix butterfly.
Decimation-in-Time Split-Radix Butterfly



The split-radix butterfly mixes radix-2 and radix-4 decompositions and is L-shaped

Further decomposition of the half- and quarter-length DFTs yields the full split-radix algorithm. The mix of different-length FFTs in different parts of the flowgraph results in a somewhat irregular algorithm; Sorensen et al. show how to adjust the computation such that the data retains the simpler radix-2 bit-reverse order. A decimation-in-frequency split-radix FFT can be derived analogously.



The split-radix transform has L-shaped butterflies

The multiplicative complexity of the split-radix algorithm is only about two-thirds that of the radix-2 FFT, and is better than the radix-4 FFT or any higher power-of-two radix as well. The additions within the complex twiddle-factor multiplies are similarly reduced, but since the underlying butterfly tree remains the same in all power-of-two algorithms, the butterfly additions remain the same and the overall reduction in additions is much less.

	Complex M/As	**Real M/As (4/2)**	**Real M/As (3/3)**
Multiplies	$O\left[\frac{N}{3}\log_2 N\right]$	$\frac{4}{3}N\log_2 N - \frac{38}{9}N + 6 + \frac{2}{9}(-1)^M$	$N\log_2 N - 3N + 4$
Additions	$O[N\log_2 N]$	$\frac{8}{3}N\log_2 N - \frac{16}{9}N + 2 + \frac{2}{9}(-1)^M$	$3N\log_2 N - 3N + 4$

Operation Counts

Comments

- The split-radix algorithm has a somewhat irregular structure. Successful progams have been written (Sorensen) for uni-processor machines, but it may be difficult to efficiently code the split-radix algorithm for vector or multi-processor machines.
- G. Bruun's algorithm requires only $N - 2$ more operations than the split-radix algorithm and has a regular structure, so it might be better for multi-processor or special-purpose hardware.
- The execution time of FFT programs generally depends more on compiler- or hardware-friendly software design than on the exact computational complexity. See Efficient FFT Algorithm and Programming Tricks for further pointers and links to good code.

Efficient FFT Algorithm and Programming Tricks

The use of FFT algorithms such as the radix-2 decimation-in-time or decimation-in-frequency methods result in tremendous savings in computations when computing the discrete Fourier transform. While most of the speed-up of FFTs comes from this, careful implementation can provide additional savings ranging from a few percent to several-fold increases in program speed.

Precompute twiddle factors

The twiddle factor, or $W_N^k = e^{-\left(i\frac{2\pi k}{N}\right)}$, terms that multiply the intermediate data in the FFT algorithms consist of cosines and sines that each take the equivalent of several multiplies to compute. However, at most N unique twiddle factors can appear in any FFT or DFT algorithm. (For example, in the radix-2 decimation-in-time FFT, only $\frac{N}{2}$ twiddle factors $\forall k, k = \left\{0, 1, 2, \ldots, \frac{N}{2} - 1\right\} : \left({W_N}^k\right)$ are used.) These twiddle factors can be precomputed once and stored in an array in computer memory, and accessed in the FFT algorithm by **table lookup**. This simple technique yields very substantial savings and is almost always used in practice.

Compiler-friendly programming

On most computers, only some of the total computation time of an FFT is spent performing the FFT butterfly computations; determining indices, loading and storing data, computing loop parameters and other operations consume the majority of cycles. Careful programming that allows the compiler to generate efficient code can make a several-fold improvement in the run-time of an FFT. The best choice of radix in terms of program speed may depend more on characteristics of the hardware (such as the number of CPU registers) or compiler than on the exact number of computations. Very often the manufacturer's library codes are carefully crafted by experts who know intimately both the hardware and compiler architecture and how to get the most performance out of them, so use of well-written FFT libraries is generally recommended. Certain freely available programs and libraries are also very good. Perhaps the best current general-purpose library is the

FFTW package; information can be found at http://www.fftw.org. A paper by Frigo and Johnson describes many of the key issues in developing compiler-friendly code.

Program in assembly language

While compilers continue to improve, FFT programs written directly in the assembly language of a specific machine are often several times faster than the best compiled code. This is particularly true for DSP microprocessors, which have special instructions for accelerating FFTs that compilers don't use. (I have myself seen differences of up to 26 to 1 in favor of assembly!) Very often, FFTs in the manufacturer's or high-performance third-party libraries are hand-coded in assembly. For DSP microprocessors, the codes developed by Meyer, Schuessler, and Schwarz are perhaps the best ever developed; while the particular processors are now obsolete, the techniques remain equally relevant today. Most DSP processors provide special instructions and a hardware design favoring the radix-2 decimation-in-time algorithm, which is thus generally fastest on these machines.

Special hardware

Some processors have special hardware accelerators or co-processors specifically designed to accelerate FFT computations. For example, AMI Semiconductor's Toccata ultra-low-power DSP microprocessor family, which is widely used in digital hearing aids, have on-chip FFT accelerators; it is always faster and more power-efficient to use such accelerators and whatever radix they prefer.

In a surprising number of applications, almost all of the computations are FFTs. A number of special-purpose chips are designed to specifically compute FFTs, and are used in specialized high-performance applications such as radar systems. Other systems, such as OFDM-based communications receivers, have special FFT hardware built into the digital receiver circuit. Such hardware can run many times faster, with much less power consumption, than FFT programs on general-purpose processors.

Effective memory management

Cache misses or excessive data movement between registers and memory can greatly slow down an FFT computation. Efficient programs such as the FFTW package are carefully designed to minimize these inefficiences. In-place algorithms reuse the data memory throughout the transform, which can reduce cache misses for longer lengths.

Real-valued FFTs

FFTs of real-valued signals require only half as many computations as with complex-valued data. There are several methods for reducing the computation, which are described in more detail in Sorensen et al.

1. Use DFT symmetry properties to do two real-valued DFTs at once with one FFT program
2. Perform one stage of the radix-2 decimation-in-time decomposition and compute the two length-$\frac{N}{2}$ DFTs using the above approach.
3. Use a direct real-valued FFT algorithm; see H.V. Sorensen et.al.

Special cases

Occasionally only certain DFT frequencies are needed, the input signal values are mostly zero, the signal is real-valued (as discussed above), or other special conditions exist for which faster algorithms can be developed. Sorensen and Burrus describe slightly faster algorithms for pruned or zero-padded data. Goertzel's algorithm is useful when only a few DFT outputs are needed. The running FFT can be faster when DFTs of highly overlapped blocks of data are needed, as in a spectrogram.

Higher-radix algorithms

Higher-radix algorithms, such as the radix-4, radix-8, or split-radix FFTs, require fewer computations and can produce modest but worthwhile savings. Even the split-radix FFT reduces the multiplications by only 33% and the additions by a much lesser amount relative to the radix-2 FFTs;

significant improvements in program speed are often due to implicit loop-unrolling or other compiler benefits than from the computational reduction itself!

Fast bit-reversal

Bit-reversing the input or output data can consume several percent of the total run-time of an FFT program. Several fast bit-reversal algorithms have been developed that can reduce this to two percent or less, including the method published by D.M.W. Evans.

Trade additions for multiplications

When FFTs first became widely used, hardware multipliers were relatively rare on digital computers, and multiplications generally required many more cycles than additions. Methods to reduce multiplications, even at the expense of a substantial increase in additions, were often beneficial. The prime factor algorithms and the Winograd Fourier transform algorithms, which required fewer multiplies and considerably more additions than the power-of-two-length algorithms, were developed during this period. Current processors generally have high-speed pipelined hardware multipliers, so trading multiplies for additions is often no longer beneficial. In particular, most machines now support single-cycle multiply-accumulate (MAC) operations, so balancing the number of multiplies and adds and combining them into single-cycle MACs generally results in the fastest code. Thus, the prime-factor and Winograd FFTs are rarely used today unless the application requires FFTs of a specific length.

It is possible to implement a complex multiply with 3 real multiplies and 5 real adds rather than the usual 4 real multiplies and 2 real adds:

$$(C + iS)\,(X + iY) = CX - SY + i\,(CY + SX)$$

but alernatively

$$Z = C\,(X - Y)$$

$$D = C + S$$

$$E = C - S$$

$$CX - SY = EY + Z$$

$$CY + SX = DX - Z$$

In an FFT, D and E come entirely from the twiddle factors, so they can be precomputed and stored in a look-up table. This reduces the cost of the complex twiddle-factor multiply to 3 real multiplies and 3 real adds, or one less and one more, respectively, than the conventional 4/2 computation.

Special butterflies

Certain twiddle factors, namely $W_N^0 = 1$, $W_N^{\frac{N}{2}}$, $W_N^{\frac{N}{4}}$, $W_N^{\frac{N}{8}}$, $W_N^{\frac{3N}{8}}$, etc., can be implemented with no additional operations, or with fewer real operations than a general complex multiply. Programs that specially implement such butterflies in the most efficient manner throughout the algorithm can reduce the computational cost by up to several N multiplies and additions in a length-N FFT.

Practical Perspective

When optimizing FFTs for speed, it can be important to maintain perspective on the benefits that can be expected from any given optimization. The following list categorizes the various techniques by potential benefit; these will be somewhat situation- and machine-dependent, but clearly one should begin with the most significant and put the most effort where the pay-off is likely to be largest.
Methods to speed up computation of DFTs

- **Tremendous Savings**

    1. FFT ($\frac{N}{\log_2 N}$ savings)

- **Substantial Savings** (*2*)

1. Table lookup of cosine/sine
2. Compiler tricks/good programming
3. Assembly-language programming
4. Special-purpose hardware
5. Real-data FFT for real data (factor of 2)
6. Special cases

- **Minor Savings**

1. radix-4, split-radix (-10% - +30%)
2. special butterflies
3. 3-real-multiplication complex multiply
4. Fast bit-reversal (up to 6%)

Note: On general-purpose machines, computation is only part of the total run time. Address generation, indexing, data shuffling, and memory access take up much or most of the cycles.

Note:A well-written radix-2 program will run much faster than a poorly written split-radix program!

Multidimensional Index Maps

Multidimensional Index Maps for DIF and DIT algorithms

Decimation-in-time algorithm

Radix-2 DIT:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk} = \sum_{n=0}^{\frac{N}{2}-1} x(2n) W_N^{2nk} + \sum_{n=0}^{\frac{N}{2}-1} x(2n+1) W_N^{(2n+1)k}$$

Formalization: Let $n = n_1 + 2n_2$: $n_1 = [0, 1]$: $n_2 = \left[0, 1, 2, \ldots, \frac{N}{2} - 1\right]$

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk} = \sum_{n_1=0}^{1} \sum_{n_2=0}^{\frac{N}{2}-1} x(n_1 + 2n_2) W_N^{(n_1+2n_2)k}$$

Also, let $k = \frac{N}{2} k_1 + k_2$: $k_1 = [0, 1]$: $k_2 = \left[0, 1, 2, \ldots, \frac{N}{2} - 1\right]$

Note: As long as there is a one-to-one correspondence between the original indices $[n, k] = [0, 1, 2, \ldots, N-1]$ and the n, k generated by the index map, the computation is the **same**; only the order in which the sums are done is changed.

Rewriting the DFT formula in terms of index map $n = n_1 + 2n_2$, $k = \frac{N}{2} k_1 + k_2$:
Equation:

Math input error

Note: Key to FFT is choosing index map so that one of the cross-terms disappears!

Exercise:

Problem: What is an index map for a radix-4 DIT algorithm?

Exercise:

Problem: What is an index map for a radix-4 DIF algortihm?

Exercise:

Problem:

What is an index map for a radix-3 DIT algorithm? (N a multiple of 3)

For arbitrary composite N $= N_1 N_2$, we can define an index map

$$n = n_1 + N_1 n_2$$

$$k = N_2 k_1 + k_2$$

$$n_1 = [0, 1, 2, \ldots, N_1 - 1]$$

$$k_1 = [0, 1, 2, \ldots, N_1 - 1]$$

$$n_2 = [0, 1, 2, \ldots, N_2 - 1]$$

$$k_2 = [0, 1, 2, \ldots, N_2 - 1]$$

Equation:

$$
\begin{aligned}
X(k) &= X(k_1, k_2) \\
&= \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x(n_1, n_2) W_N^{N_2 n_1 k_1} W_N^{n_1 k_2} W_N^{N k_1 n_2} W_N^{N_1 n_2 k_2} \\
&= \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x(n_1, n_2) W_{N_1}^{n_1 k_1} W_N^{n_1 k_2} 1 W_{N_2}^{n_2 k_2} \\
&= \mathrm{DFT}_{n_1, N_1} \left[W_N^{n_1 k_2} \, \mathrm{DFT}_{n_2, N_2} \left[x(n_1, n_2) \right] \right]
\end{aligned}
$$

Computational cost in multipliesl "Common Factor Algorithm (CFA)"

- N_1 length-N_2 DFTs ⇒ ${N_1 N_2}^2$
- N twiddle factors ⇒ N
- N_2 length-N_1 DFTs ⇒ ${N_2 N_1}^2$
- **Total** ${N_1 N_2}^2 + N_1 N_2 + {N_2 N_1}^2 = N\left(N_1 + N_2 + 1\right)$

"Direct": $N^2 = N\left(N_1 N_2\right)$

Example:

$$N_1 = 16$$

$$N_2 = 15$$

$$N = 240$$

$$\text{direct} = 240^2 = 57600$$

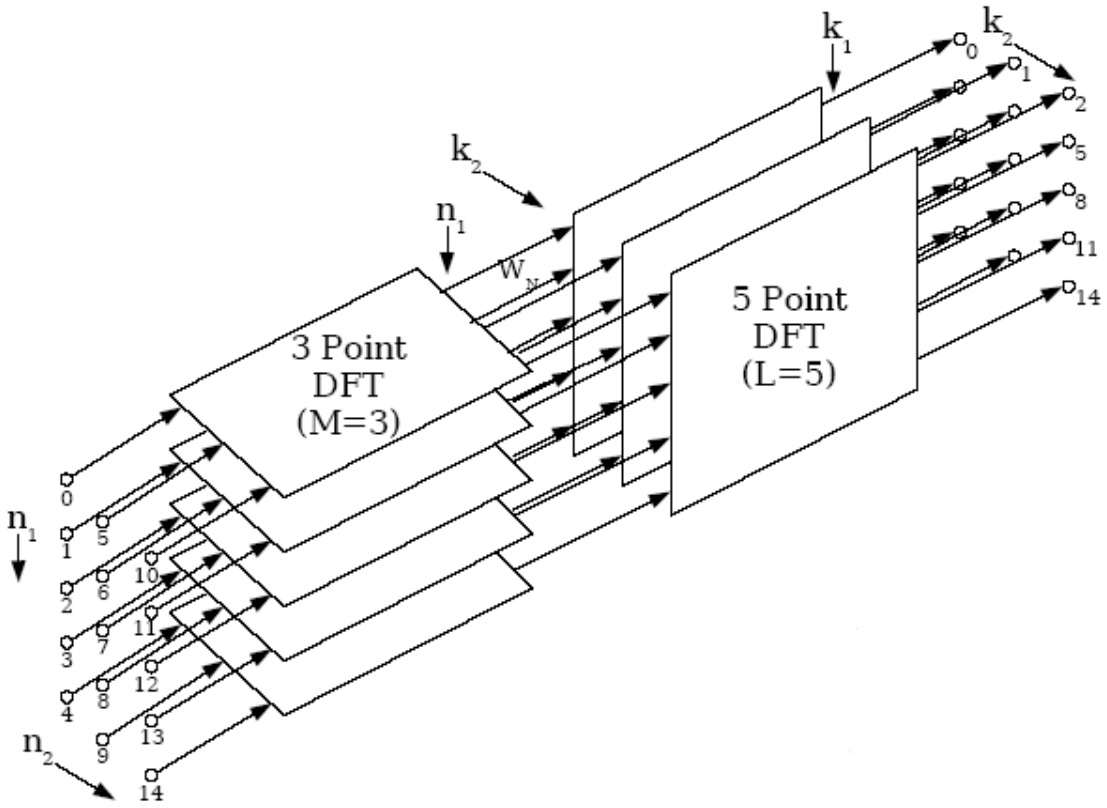
$$\text{CFA} = 7680$$

Tremendous saving for **any** composite N

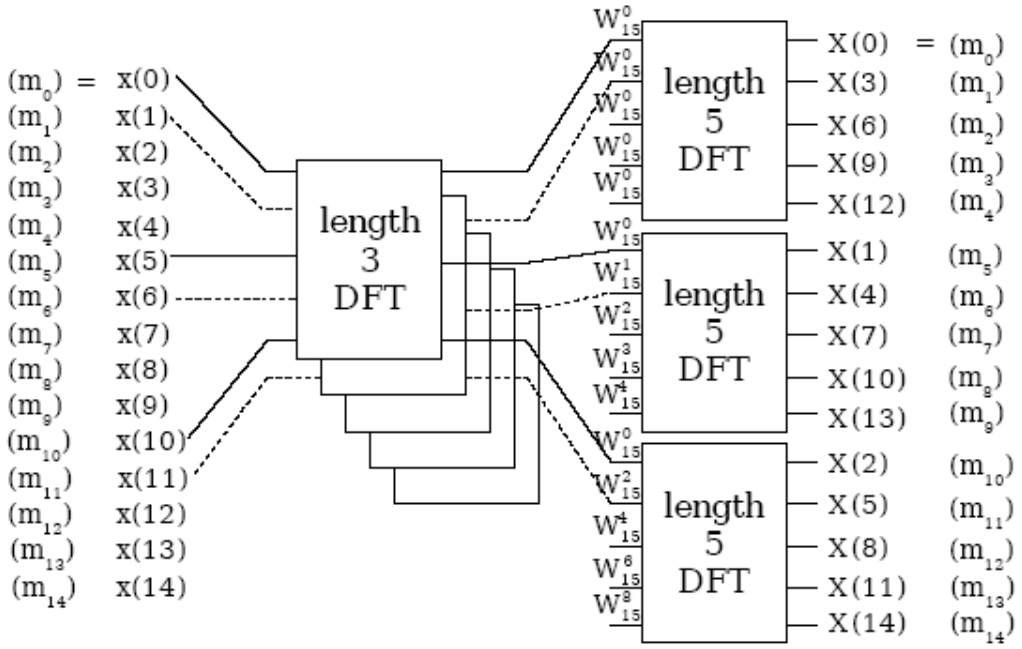
Pictorial Representations

$$n = n_1 + 5n_2,\, k = 3k_1 + k_2$$

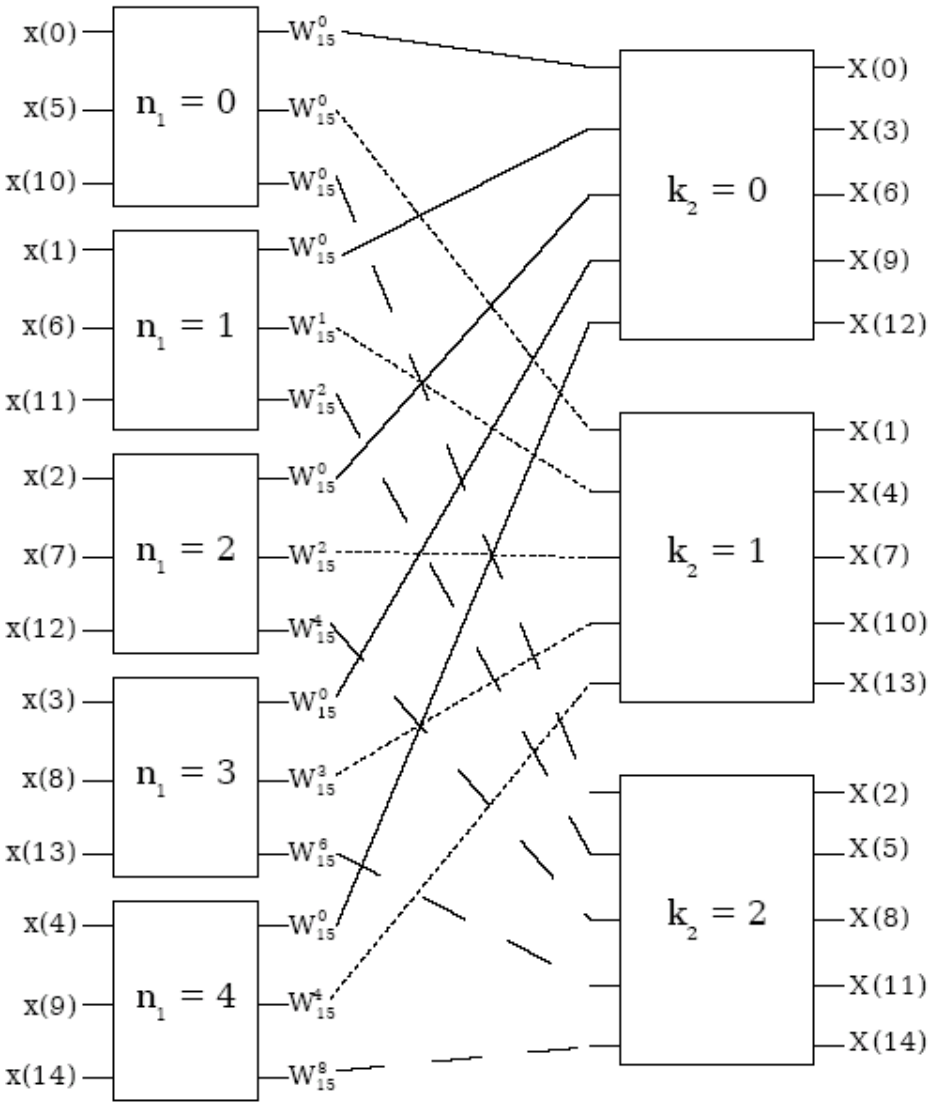
Emphasizes Multi-dimensional structure



Emphasizes computer memory organization


(m0) = x(0)
(m1) x(1)
(m2) x(2)
(m3) x(3)
(m4) x(4)
(m5) x(5)
(m6) x(6)
(m7) x(7)
(m8) x(8)
(m9) x(9)
(m10) x(10)
(m11) x(11)
(m12) x(12)
(m13) x(13)
(m14) x(14)
length 3 DFT
W0 15
W0 15
W0 15
W0 15
W0 15
length 5 DFT
X(0) = (m0)
X(3) (m1)
X(6) (m2)
X(9) (m3)
X(12) (m4)
W0 15
W1 15
W2 15
W3 15
W4 15
length 5 DFT
X(1) (m5)
X(4) (m6)
X(7) (m7)
X(10) (m8)
X(13) (m9)
W0 15
W2 15
W4 15
W6 15
W8 15
length 5 DFT
X(2) (m10)
X(5) (m11)
X(8) (m12)
X(11) (m13)
X(14) (m14)

Easy to draw



Exercise:

Problem: Can the composite CFAs be implemented in-place?

Exercise:

Problem: What do we do with $N = N_1 N_2 N_3$?

The Prime Factor Algorithm

General Index Maps

$$n = (K_1 n_1 + K_2 n_2) \bmod N$$

$$n = (K_3 k_1 + K_4 k_2) \bmod N$$

$$n_1 = [0, 1, \ldots, N_1 - 1]$$

$$k_1 = [0, 1, \ldots, N_1 - 1]$$

$$n_2 = [0, 1, \ldots, N_2 - 1]$$

$$k_2 = [0, 1, \ldots, N_2 - 1]$$

The basic ideas is to simply **reorder** the DFT computation to expose the redundancies in the DFT, and exploit these to reduce computation!

Three conditions must be satisfied to make this map serve our purposes

1. Each map must be one-to-one from 0 to $N - 1$, because we want to do the **same** computation, just in a different order.
2. The map must be cleverly chosen so that computation is reduced
3. The map should be chosen to make the short-length transforms be DFTs. (Not essential, since fast algorithms for short-length DFT-like computations could be developed, but it makes our work easier.)

Conditions for one-to-oneness of general index map

Case I

N_1, N_2 relatively prime (greatest common denominator 1) i.e. $\gcd(N_1, N_2) = 1$

$K_1 = aN_2$ and/or $K_2 = bN_1$ and $\gcd(K_1, N_1) = 1$, $\gcd(K_2, N_2) = 1$

Case II

N_1, N_2 **not** relatively prime: $\gcd(N_1, N_2) > 1$

$K_1 = aN_2$ and $K_2 \neq bN_1$ and gcd $(a, N_1) = 1$, gcd $(K_2, N_2) = 1$ or $K_1 \neq aN_2$ and $K_2 = bN_1$ and gcd $(K_1, N_1) = 1$, gcd $(b, N_2) = 1$ where K_1, K_2, K_3, K_4, N_1, N_2, a, b integers

Note: Requires number-theory/abstract-algebra concepts. Reference: C.S. Burrus

Note: Conditions of one-to-oneness must apply to both k and n

Conditions for arithmetic savings

Equation:

$$\begin{aligned} X(k_1, k_2) &= \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x(n_1, n_2) W_N^{(K_1 n_1 + K_2 n_2)(K_3 k_1 + K_4 k_2} \\ &= \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x(n_1, n_2) W_N^{K_1 K_3 n_1 k_1} W_N^{K_1 K_4 n_1 k_2} W_N^{K_2 K_3 n_2 k_1} W_N^{K_2 K_4 n_2 k_2} \end{aligned}$$

- $K_1 K_4 \bmod N = 0$ exclusive or $K_2 K_3 \bmod N = 0 \Rightarrow$ Common Factor Algorithm (CFA). Then

$$X(k) = \text{DFT}_{Ni} \left[\text{twiddle factors} \, \text{DFT}_{Nj} \left[x(n_1, n_2)\right]\right]$$

- $K_1 K_4 \bmod N$ **and** $K_2 K_3 \bmod N = 0 \Rightarrow$ Prime Factor Algorithm (PFA).

$$X(k) = \text{DFT}_{Ni} \left[\, \text{DFT}_{Nj} \,\right]$$

  No twiddle factors!

Note: A PFA exists only and always for relatively prime N_1, N_2

Conditions for short-length transforms to be DFTs

$K_1 K_3 \bmod N = N_2$ and $K_2 K_4 \bmod N = N_1$

Note: Convenient choice giving a PFA

$K_1 = N_2$, $K_2 = N_1$, $K_3 = N_2 {N_2}^{-1} \bmod N_1 \bmod N_1$,
$K_4 = N_1 {N_1}^{-1} \bmod N_2 \bmod N_2$ where ${N_1}^{-1} \bmod N_2$ is an integer such that $N_1 {N_1}^{-1} \bmod = 1$

Example:
$N_1 = 3$, $N_2 = 5$ $N = 15$

$$\mathrm{n} = (5n_1 + 3n_2) \bmod 15$$

$$k = (10k_1 + 6k_2) \bmod 15$$

1. **Checking Conditions for one-to-oneness**

$$5 = K_1 = aN_2 = 5a$$

$$3 = K_2 = bN_1 = 3b$$

$$\gcd(5, 3) = 1$$

$$\gcd(3, 5) = 1$$

$$10 = K_3 = aN_2 = 5a$$

$$6 = K_4 = bN_1 = 3b$$

$$\gcd(10, 3) = 1$$

$$\gcd(6, 5) = 1$$

2. **Checking conditions for reduced computation**

$$K_1 K_4 \bmod 15 = 5 \times 6 \bmod 15 = 0$$

$$K_2 K_3 \bmod 15 = 3 \times 10 \bmod 15 = 0$$

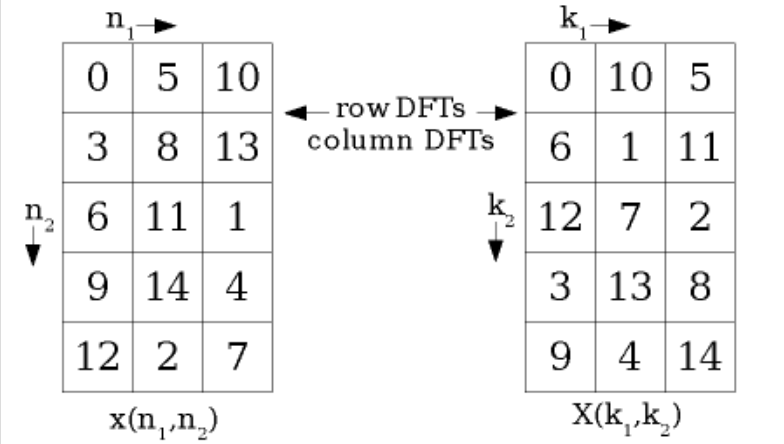
3. **Checking Conditions for making the short-length transforms be DFTS**

$$K_1 K_3 \bmod 15 = 5 \times 10 \bmod 15 = 5 = N_2$$

$$K_2 K_4 \bmod 15 = 3 \times 6 \bmod 15 = 3 = N_1$$

Therefore, this is a prime factor map.

2-D map



$$n = (5n_1 + 3n_2) \bmod 15 \text{ and}$$
$$k = (10k_1 + 6k_2) \bmod 15$$

Operation Counts

- N_2 length- N_1 DFTs N_1 length- N_2 DFTs

  $$N_2 {N_1}^2 + N_1 {N_2}^2 = N\left(N_1 + N_2\right)$$

  complex multiplies
- Suppose $N = N_1 N_2 N_3 \ldots N_{\mathrm{M}}$

  $$N\left(N_1 + N_2 + \ldots + N_M\right)$$

  Complex multiplies

Note: radix-2, radix-4 eliminate all multiplies in short-length DFTs, but have twiddle factors: PFA eliminates all twiddle factors, but ends up with multiplies in short-length DFTs. Surprisingly, total operation counts end up being very similar for similar lengths.

Fast Convolution

Fast Circular Convolution

Since,

$$\sum_{m=0}^{N-1} x(m)h(n-m) \bmod N = y(n) \text{is equivalent to} Y(k) = X(k)H(k)$$

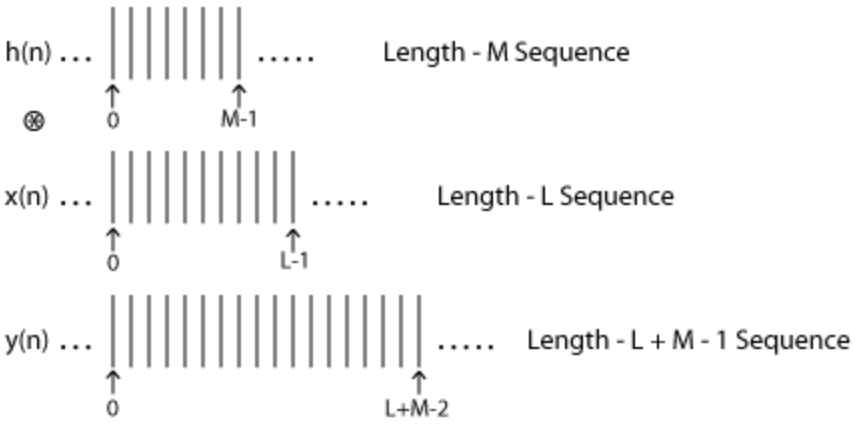
$y(n)$ can be computed as $y(n) = \text{IDFT}\left[\text{DFT}\left[x(n)\right]\text{DFT}\left[h(n)\right]\right]$
Cost

- **Direct**
  - N^2 complex multiplies.
  - $N\left(N-1\right)$ complex adds.
- **Via FFTs**
  - 3 FFTs + N multipies.
  - $N + \frac{3N}{2}\log_2 N$ complex multiplies.
  - $3\left(N\log_2 N\right)$ complex adds.

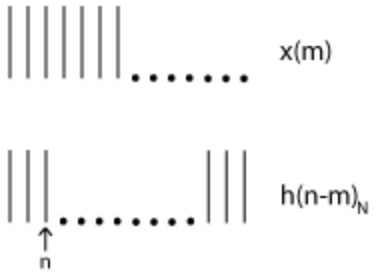
If $H(k)$ can be precomputed, cost is only 2 FFts + N multiplies.

Fast Linear Convolution

DFT produces cicular convolution. For linear convolution, we must zero-pad sequences so that circular wrap-around always wraps over zeros.



To achieve linear convolution using fast circular convolution, we must use zero-padded DFTs of length $N \geq L + M - 1$



Choose shortest convenient N (usually smallest power-of-two greater than or equal to $L + M - 1$)

$$y(n) = \mathrm{IDFT}_N \left[\mathrm{DFT}_N \left[x(n) \right] \mathrm{DFT}_N \left[h(n) \right] \right]$$

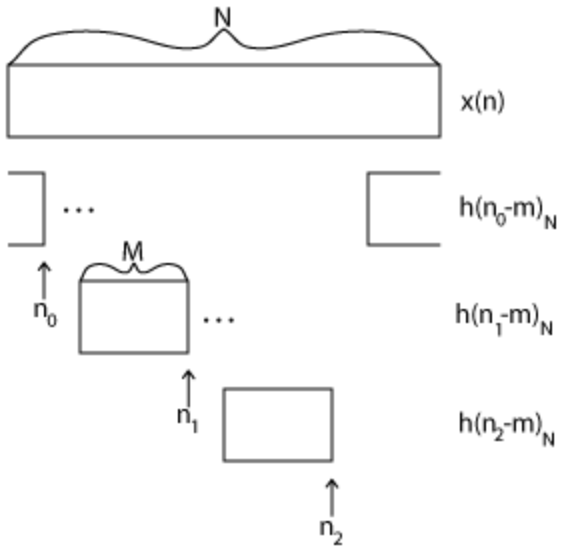
Note: There is some inefficiency when compared to circular convolution due to longer zero-padded DFTs. Still, $O\left(\frac{N}{\log_2 N}\right)$ savings over direct computation.

Running Convolution

Suppose $L = \infty$, as in a real time filter application, or $L \gg M$. There are efficient block methods for computing fast convolution.

Overlap-Save (OLS) Method

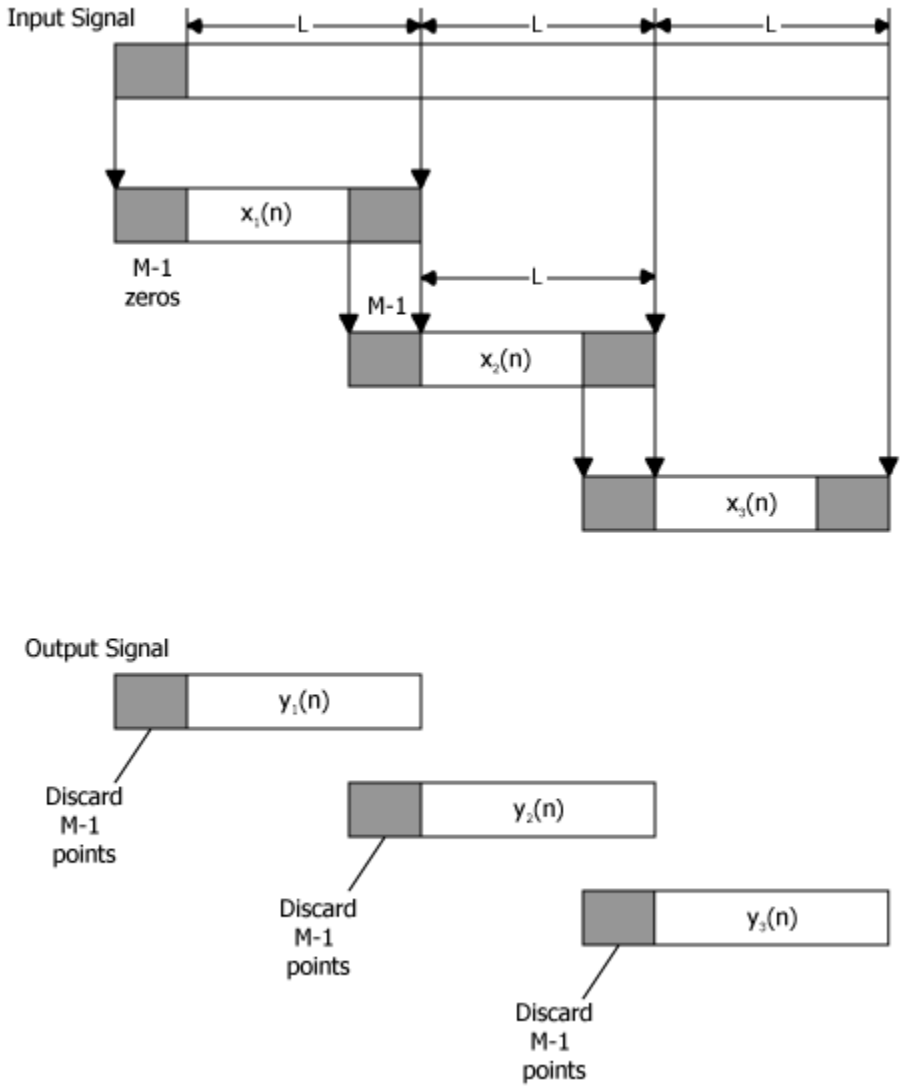
Note that if a length-M filter $h(n)$ is circularly convulved with a length-N segment of a signal $x(n)$,



the first $M - 1$ samples are wrapped around and thus is **incorrect**. However, for $M - 1 \leq n \leq N - 1$,the convolution is linear convolution, so these samples are correct. Thus $N - M + 1$ good outputs are produced for each length-N circular convolution.

The Overlap-Save Method: Break long signal into successive blocks of N samples, each block overlapping the previous block by $M - 1$ samples. Perform circular convolution of each block with filter $h(m)$. Discard first

$M-1$ points in each output block, and concatenate the remaining points to create $y(n)$.



Computation cost for a length-N equals 2^n FFT per output sample is (assuming precomputed $H(k)$) 2 FFTs and N multiplies

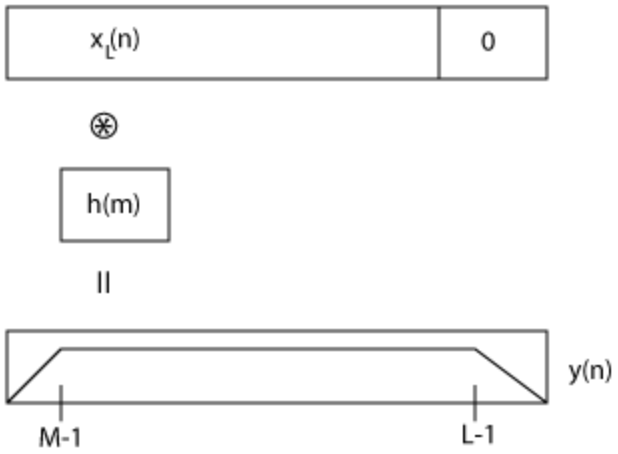
$$\frac{2\left(\frac{N}{2}\log_2 N\right)+N}{N-M+1} = \frac{N\left(\log_2 N+1\right)}{N-M+1} \text{complex multiplies}$$

$$\frac{2\left(N\log_2 N\right)}{N-M+1} = \frac{2N\log_2 N}{N-M+1}\text{complex adds}$$

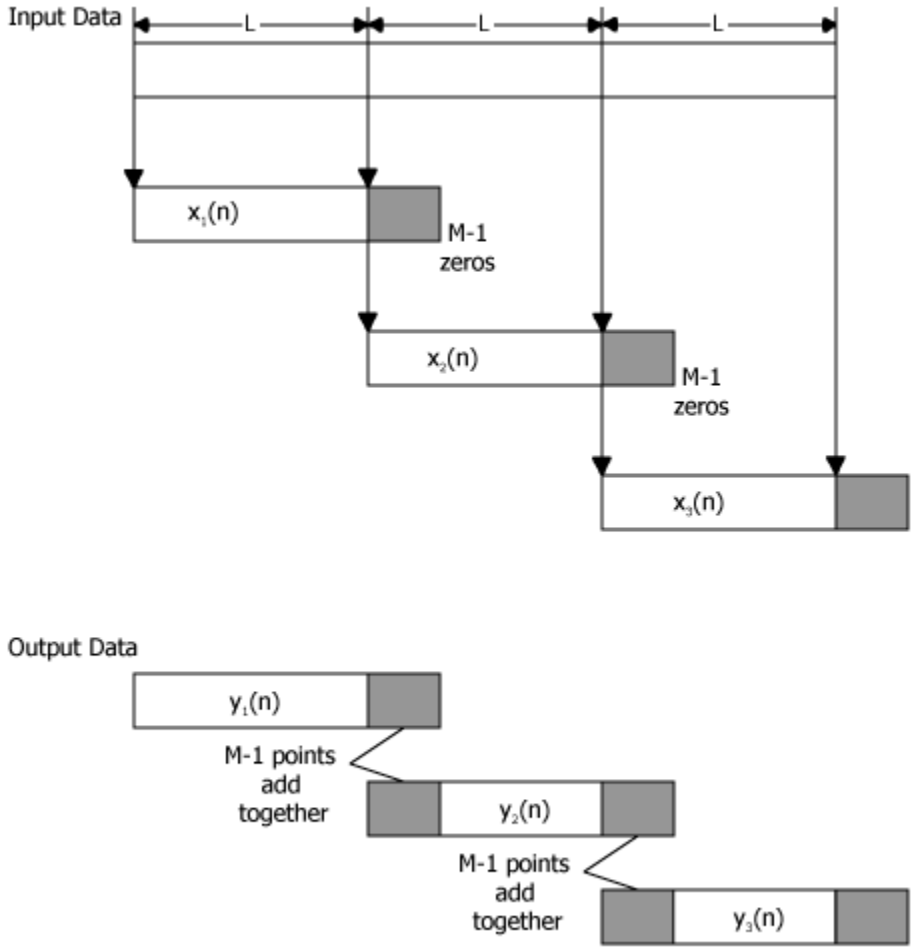
Compare to M mults, $M-1$ adds per output point for direct method. For a given M, optimal N can be determined by finding N minimizing operation counts. Usualy, optimal N is $4M \leq N_{\text{opt}} \leq 8M$.

Overlap-Add (OLA) Method

Zero-pad length-L blocks by $M-1$ samples.



Add successive blocks, overlapped by $M-1$ samples, so that the tails sum to produce the complete linear convolution.



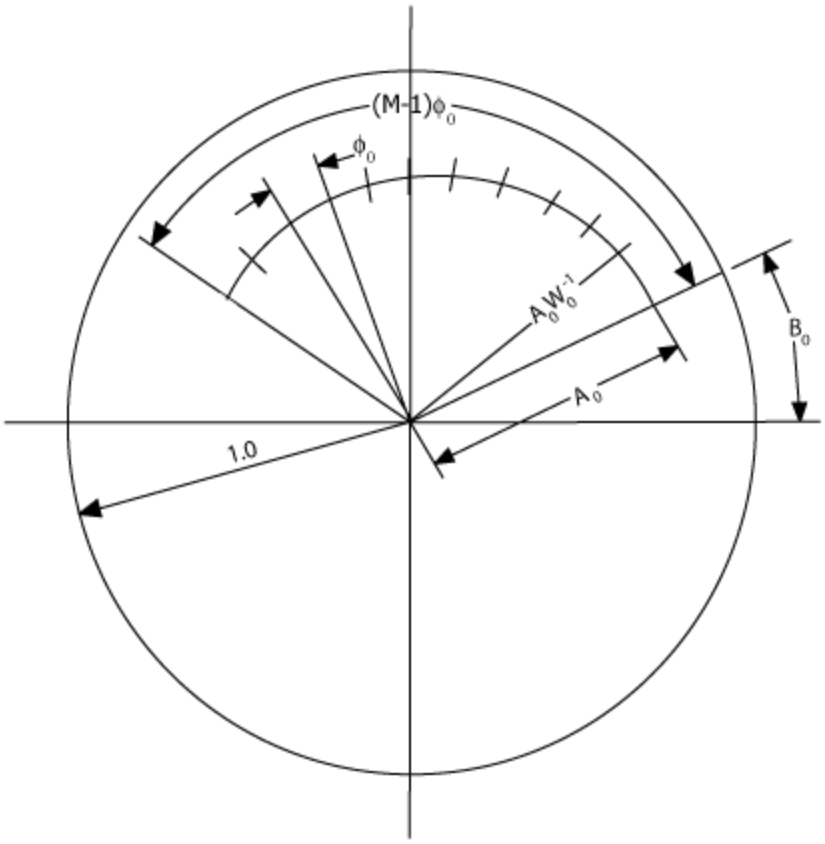
Computational Cost: Two length $N = L + M - 1$ FFTs and M mults and $M - 1$ adds per L output points; essentially the sames as OLS method.

Chirp-z Transform

Let $z^k = AW^{-k}$, where $A = A_o e^{i\theta_o}$, $W = W_o e^{-(i\varphi_o)}$.

We wish to compute M samples, $k = [0, 1, 2, \ldots, M-1]$ of

$$X(z_k) = \sum_{n=0}^{N-1} x(n) {z_k}^{-n} = \sum_{n=0}^{N-1} x(n) A^{-n} W^{\mathrm{nk}}$$



Note that
$\left(\left(k-n\right)^2 = n^2 - 2nk + k^2\right) \Rightarrow \left(nk = \frac{1}{2}\left(n^2 + k^2 - \left(k-n\right)^2\right)\right)$, So

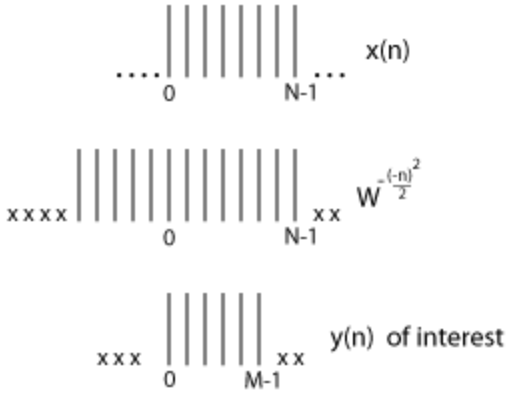
$$X(z_k) = \sum_{n=0}^{N-1} x(n) A^{-n} W^{\frac{n^2}{2}} W^{\frac{k^2}{2}} W^{\frac{-(k-n)^2}{2}}$$

$$W^{\frac{k^2}{2}} \sum_{n=0}^{N-1} x(n) A^{-n} W^{\frac{n^2}{2}} W^{\frac{-(k-n)^2}{2}}$$

Thus, $X(z_k)$ can be compared by

1. Premultiply $x(n)$ by $A^n W^{\frac{n^2}{2}}$, $n = [0, 1, \ldots, N-1]$ to make $y(n)$
2. Linearly convolve with $W^{\frac{-(k-n)^2}{2}}$
3. Post multiply by to get $W^{\frac{k^2}{2}}$ to get $X(z_k)$.

1. and 3. require N and M operations respectively. 2. can be performed efficiently using fast convolution.



$W^{-\frac{n^2}{2}}$ is required only for $-(N-1) \leq n \leq M-1$, so this linear convolution can be implemented with $L \geq N + M - 1$ FFTs.

> **Note:** Wrap $W^{-\frac{n^2}{2}}$ around L when implementing with circular convolution.

So, a weird-length DFT can be implemented relatively efficiently using power-of-two algorithms via the chirp-z transform.

Also useful for "zoom-FFTs".

FFTs of prime length and Rader's conversion

The power-of-two FFT algorithms, such as the radix-2 and radix-4 FFTs, and the common-factor and prime-factor FFTs, achieve great reductions in computational complexity of the DFT when the length, N, is a composite number. DFTs of prime length are sometimes needed, however, particularly for the short-length DFTs in common-factor or prime-factor algorithms. The methods described here, along with the composite-length algorithms, allow fast computation of DFTs of **any** length.

There are two main ways of performing DFTs of prime length:

1. Rader's conversion, which is most efficient, and the
2. Chirp-z transform, which is simpler and more general.

Oddly enough, both work by turning prime-length DFTs into convolution! The resulting convolutions can then be computed efficiently by either

1. fast convolution via composite-length FFTs (simpler) or by
2. Winograd techniques (more efficient)

Rader's Conversion

Rader's conversion is a one-dimensional index-mapping scheme that turns a length-N DFT (N prime) into a length-($N-1$) convolution and a few additions. Rader's conversion works **only** for prime-length N.

An **index map** simply rearranges the order of the sum operation in the DFT definition. Because addition is a commutative operation, the same mathematical result is produced from any order, as long as all of the same terms are added once and only once. (This is the condition that defines an index map.) Unlike the multi-dimensional index maps used in deriving common factor and prime-factor FFTs, Rader's conversion uses a one-dimensional index map in a finite group of N integers: $k = r^m \bmod N$

Fact from number theory

If N is prime, there exists an integer "r" called a **primitive root**, such that the index map $k = r^m \bmod N$, $m = [0, 1, 2, \ldots, N-2]$, uniquely generates all elements $k = [1, 2, 3, \ldots, N-1]$

Example:
$N = 5$, $r = 2$

$$2^0 \bmod 5 = 1$$

$$2^1 \bmod 5 = 2$$

$$2^2 \bmod 5 = 4$$

$$2^3 \bmod 5 = 3$$

Another fact from number theory

For N prime, the inverse of r (i.e. $r^{-1}r \bmod N = 1$ is also a primitive root (call it r^{-1}).

Example:
$N = 5$, $r = 2$ $r^{-1} = 3$

$$2 \times 3 \bmod 5 = 1$$

$$3^0 \bmod 5 = 1$$

$$3^1 \bmod 5 = 3$$

$$3^2 \bmod 5 = 4$$

$$3^3 \bmod 5 = 2$$

So why do we care? Because we can use these facts to turn a DFT into a convolution!

Rader's Conversion

Let
$\forall mn, (m = [0, 1, \ldots, N-2]) \ \wedge \ n \in [1, 2, \ldots, N-1] : (n = r^{-m} \bmod N)$
, $\forall pk, (p = [0, 1, \ldots, N-2]) \ \wedge \ k \in [1, 2, \ldots, N-1] : (k = r^{p} \bmod N)$

$$X(k) = \sum_{n=0}^{N-1} x(\mathrm{n}) W_N^{nk} = \begin{cases} x(0) + \sum_{n=1}^{N-1} x(n) W_N^{nk} & \text{if } \ k \neq 0 \\ \sum_{n=0}^{N-1} x(n) & \text{if } \ k = 0 \end{cases}$$

where for convenience $W_N^{nk} = e^{-\left(i\frac{2\pi nk}{N}\right)}$ in the DFT equation. For $k \neq 0$
Equation:

$$\begin{aligned} X(r^p \bmod N) &= \sum_{m=0}^{N-2} x(r^{-m} \bmod N) W^{r^p r^{-m}} + x(0) \\ &= \sum_{m=0}^{N-2} x(r^{-m} \bmod N) W^{r^{p-m}} + x(0) \\ &= x(0) + x\left(r^{-l} \bmod N\right) {*} W^{r^l} \end{aligned}$$

where $l = [0, 1, \ldots, N-2]$

Example:
$N = 5, r = 2, r^{-1} = 3$

$$\begin{pmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \\ X(4) \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 & 4 \\ 0 & 2 & 4 & 1 & 3 \\ 0 & 3 & 1 & 4 & 2 \\ 0 & 4 & 3 & 2 & 1 \end{pmatrix} \begin{pmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \\ x(4) \end{pmatrix}$$

$$\begin{pmatrix} X(0) \\ X(1) \\ X(2) \\ X(4) \\ X(3) \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 3 & 4 & 2 \\ 0 & 2 & 1 & 3 & 4 \\ 0 & 4 & 2 & 1 & 1 \\ 0 & 3 & 4 & 2 & 3 \end{pmatrix} \begin{pmatrix} x(0) \\ x(1) \\ x(3) \\ x(4) \\ x(2) \end{pmatrix}$$

where for visibility the matrix entries represent only the **power**, m of the corresponding DFT term W_N^m Note that the 4-by-4 circulant matrix

$$\begin{pmatrix} 1 & 3 & 4 & 2 \\ 2 & 1 & 3 & 4 \\ 4 & 2 & 1 & 1 \\ 3 & 4 & 2 & 3 \end{pmatrix}$$

corresponds to a length-4 circular convolution.

Rader's conversion turns a prime-length DFT into a few adds and a **composite-length** $(N-1)$ circular convolution, which can be computed efficiently using either

1. fast convolution via FFT and IFFT
2. index-mapped convolution algorithms and short Winograd convolution alogrithms. (Rather complicated, and trades fewer multiplies for many more adds, which may not be worthwile on most modern processors.) See R.C. Agarwal and J.W. Cooley

Winograd minimum-multiply convolution and DFT algorithms

S. Winograd has proved that a length-N circular or linear convolution or DFT requires less than $2N$ multiplies (for real data), or $4N$ real multiplies for complex data. (This doesn't count multiplies by rational fractions, like 3 or $\frac{1}{N}$ or $\frac{5}{17}$, which can be computed with additions and one overall scaling factor.) Furthermore, Winograd showed how to construct algorithms achieving these counts. Winograd prime-length DFTs and convolutions have the following characteristics:

1. Extremely efficient for small N ($N < 20$)
2. The number of adds becomes **huge** for large N.

Thus Winograd's minimum-multiply FFT's are useful only for small N. They are very important for Prime-Factor Algorithms, which generally use Winograd modules to implement the short-length DFTs. Tables giving the multiplies and adds necessary to compute Winograd FFTs for various lengths can be found in C.S. Burrus (1988). Tables and FORTRAN and TMS32010 programs for these short-length transforms can be found in C.S. Burrus and T.W. Parks (1985). The theory and derivation of these algorithms is quite elegant but requires substantial background in number theory and abstract algebra. Fortunately for the practitioner, all of the short algorithms one is likely to need have already been derived and can simply be looked up without mastering the details of their derivation.

Winograd Fourier Transform Algorithm (WFTA)

The Winograd Fourier Transform Algorithm (WFTA) is a technique that recombines the short Winograd modules in a prime-factor FFT into a composite-N structure with fewer multiplies but more adds. While theoretically interesting, WFTAs are complicated and different for every length, and on modern processors with hardware multipliers the trade of multiplies for many more adds is very rarely useful in practice today.

Choosing the Best FFT Algorithm

Choosing an FFT length

The most commonly used FFT algorithms **by far** are the power-of-two-length FFT algorithms. The Prime Factor Algorithm (PFA) and Winograd Fourier Transform Algorithm (WFTA) require somewhat fewer multiplies, but the overall difference usually isn't sufficient to warrant the extra difficulty. This is particularly true now that most processors have single-cycle pipelined hardware multipliers, so the total operation count is more relevant. As can be seen from the following table, for similar lengths the split-radix algorithm is comparable in total operations to the Prime Factor Algorithm, and is considerably better than the WFTA, although the PFA and WTFA require fewer multiplications and more additions. Many processors now support single cycle multiply-accumulate (MAC) operations; in the power-of-two algorithms all multiplies can be combined with adds in MACs, so the number of additions is the most relevant indicator of computational cost.

	FFT length	**Multiplies (real)**	**Adds(real)**	**Mults + Adds**
Radix 2	1024	10248	30728	40976
Split Radix	1024	7172	27652	34824
Prime Factor Alg	1008	5804	29100	34904

	FFT length	Multiplies (real)	Adds(real)	Mults + Adds
Winograd FT Alg	1008	3548	34416	37964

Representative FFT Operation Counts

The Winograd Fourier Transform Algorithm is particularly difficult to program and is rarely used in practice. For applications in which the transform length is somewhat arbitrary (such as fast convolution or general spectrum analysis), the length is usually chosen to be a power of two. When a particular length is required (for example, in the USA each carrier has exactly 416 frequency channels in each band in the AMPS cellular telephone standard), a Prime Factor Algorithm for all the relatively prime terms is preferred, with a Common Factor Algorithm for other non-prime lengths. Winograd's short-length modules should be used for the prime-length factors that are not powers of two. The chirp z-transform offers a universal way to compute any length DFT (for example, Matlab reportedly uses this method for lengths other than a power of two), at a few times higher cost than that of a CFA or PFA optimized for that specific length. The chirp z-transform, along with Rader's conversion, assure us that algorithms of $O(N \log N)$ complexity exist for **any** DFT length N.

Selecting a power-of-two-length algorithm

The choice of a power-of-two algorithm may not just depend on computational complexity. The latest extensions of the split-radix algorithm offer the lowest known power-of-two FFT operation counts, but the 10%-30% difference may not make up for other factors such as regularity of structure or data flow, FFT programming tricks, or special hardware features. For example, the decimation-in-time radix-2 FFT is the fastest FFT on Texas Instruments' TMS320C54x DSP microprocessors, because this processor family has special assembly-language instructions that accelerate this particular algorithm. On other hardware, radix-4 algorithms

may be more efficient. Some devices, such as AMI Semiconductor's Toccata ultra-low-power DSP microprocessor family, have on-chip FFT accelerators; it is always faster and more power-efficient to use these accelerators and whatever radix they prefer. For fast convolution, the decimation-in-frequency algorithms may be preferred because the bit-reversing can be bypassed; however, most DSP microprocessors provide zero-overhead bit-reversed indexing hardware and prefer decimation-in-time algorithms, so this may not be true for such machines. Good, compiler- or hardware-friendly programming always matters more than modest differences in raw operation counts, so manufacturers' or good third-party FFT libraries are often the best choice. The module FFT programming tricks references some good, free FFT software (including the FFTW package) that is carefully coded to be compiler-friendly; such codes are likely to be considerably faster than codes written by the casual programmer.

Multi-dimensional FFTs

Multi-dimensional FFTs pose additional possibilities and problems. The orthogonality and separability of multi-dimensional DFTs allows them to be efficiently computed by a series of one-dimensional FFTs along each dimension. (For example, a two-dimensional DFT can quickly be computed by performing FFTs of each row of the data matrix followed by FFTs of all columns, or vice-versa.) **Vector-radix FFTs** have been developed with higher efficiency per sample than row-column algorithms. Multi-dimensional datasets, however, are often large and frequently exceed the cache size of the processor, and excessive cache misses may increase the computational time greatly, thus overwhelming any minor complexity reduction from a vector-radix algorithm. Either vector-radix FFTs must be carefully programmed to match the cache limitations of a specific processor, or a row-column approach should be used with matrix transposition in between to ensure data locality for high cache utilization throughout the computation.

Few time or frequency samples

FFT algorithms gain their efficiency through intermediate computations that can be reused to compute many DFT frequency samples at once. Some applications require only a handful of frequency samples to be computed; when that number is of order less than $O(\log N)$, direct computation of those values via Goertzel's algorithm is faster. This has the additional advantage that any frequency, not just the equally-spaced DFT frequency samples, can be selected. Sorensen and Burrus developed algorithms for when most input samples are zero or only a block of DFT frequencies are needed, but the computational cost is of the same order.

Some applications, such as time-frequency analysis via the short-time Fourier transform or spectrogram, require DFTs of overlapped blocks of discrete-time samples. When the step-size between blocks is less than $O(\log N)$, the running FFT will be most efficient. (Note that any window must be applied via frequency-domain convolution, which is quite efficient for sinusoidal windows such as the Hamming window.) For step-sizes of $O(\log N)$ or greater, computation of the DFT of each successive block via an FFT is faster.

Overview of Digital Filter Design

Advantages of FIR filters

1. Straight forward conceptually and simple to implement
2. Can be implemented with fast convolution
3. Always stable
4. Relatively insensitive to quantization
5. Can have linear phase (same time delay of all frequencies)

Advantages of IIR filters

1. Better for approximating analog systems
2. For a given magnitude response specification, IIR filters often require much less computation than an equivalent FIR, particularly for narrow transition bands

Both FIR and IIR filters are very important in applications.

Generic Filter Design Procedure

1. Choose a desired response, based on application requirements
2. Choose a filter class
3. Choose a quality measure
4. Solve for the filter in class 2 optimizing criterion in 3

Perspective on FIR filtering

Most of the time, people do L^∞ optimal design, using the Parks-McClellan algorithm. This is probably the second most important technique in "classical" signal processing (after the Cooley-Tukey (radix-2) FFT).

Most of the time, FIR filters are designed to have linear phase. The most important advantage of FIR filters over IIR filters is that they can have exactly linear phase. There are advanced design techniques for minimum-phase filters, constrained L^2 optimal designs, etc. (see chapter 8 of text). However, if only the **magnitude** of the response is important, IIR filers usually require much fewer operations and are typically used, so the bulk of FIR filter design work has concentrated on linear phase designs.

Linear Phase Filters

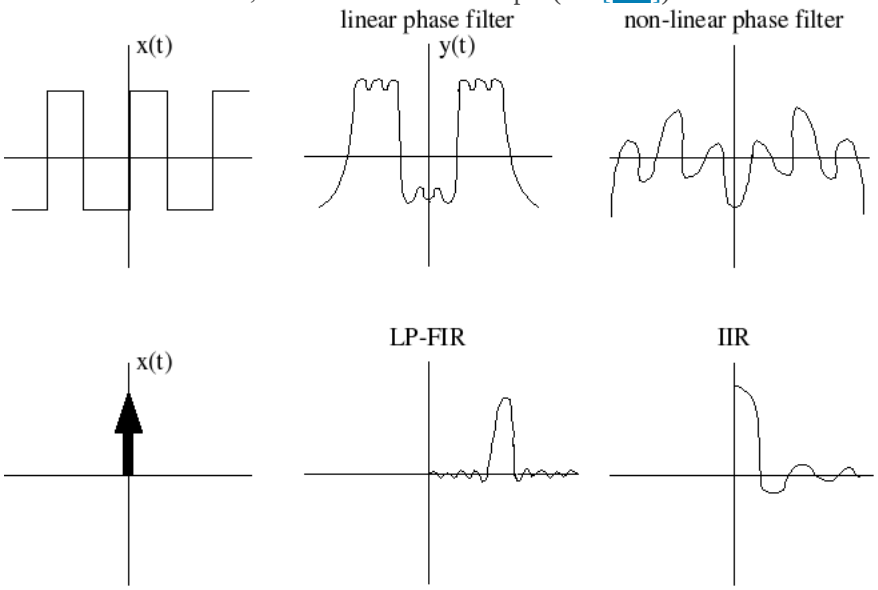
In general, for $-\pi \leq \omega \leq \pi$

$$H(\omega) = |H(\omega)|e^{-(i\theta(\omega))}$$

Strictly speaking, we say $H(\omega)$ is linear phase if

$$H(\omega) = |H(\omega)|e^{-(i\omega K)}e^{-(i\theta_0)}$$

Why is this important? A linear phase response gives the **same time delay for ALL frequencies**! (Remember the shift theorem.) This is very desirable in many applications, particularly when the appearance of the time-domain waveform is of interest, such as in an oscilloscope. (see [link])



Restrictions on h(n) to get linear phase

Equation:

$$\begin{aligned} H(\omega) &= \textstyle\sum_{h=0}^{M-1} h(n)e^{-(i\omega n)} \\ &= h(0) + h(1)e^{-(i\omega)} + h(2)e^{-(i2\omega)} + \ldots + h(M-1)e^{-(i\omega(M-1))} \\ &= e^{-(i\omega\frac{M-1}{2})}\left(h(0)e^{i\omega\frac{M-1}{2}} + \ldots + h(M-1)e^{-(i\omega\frac{M-1}{2})}\right) \\ &= e^{-(i\omega\frac{M-1}{2})}\left((h(0)+h(M-1))\cos\left(\tfrac{M-1}{2}\omega\right) + (h(1)+h(M-2))\cos\left(\tfrac{M-3}{2}\omega\right) + \ldots + i\left(h(0)\sin(\right.\right. \end{aligned}$$

For linear phase, we require the right side of [link] to be $e^{-(i\theta_0)}$(real,positive function of ω). For $\theta_0 = 0$, we thus require

$$h(0) + h(M-1) = \text{real number}$$

$$h(0) - h(M-1) = \text{pure imaginary number}$$

$$h(1) + h(M-2) = \text{pure real number}$$

$$h(1) - h(M-2) = \text{pure imaginary number}$$

$$\vdots$$

Thus $h(k) = h^*(M-1-k)$ is a **necessary** condition for the right side of [link] to be real valued, for $\theta_0 = 0$.

For $\theta_0 = \frac{\pi}{2}$, or $e^{-(i\theta_0)} = -i$, we require

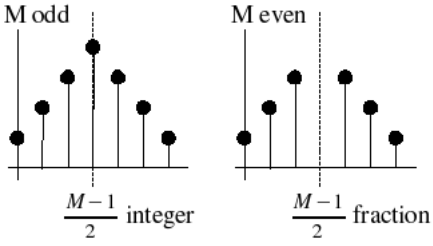
$$h(0) + h(M-1) = \text{pure imaginary}$$

$$h(0) - h(M-1) = \text{pure real number}$$

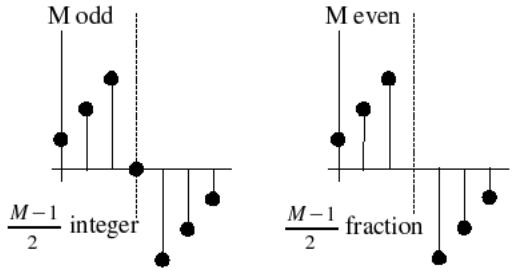
$$\vdots$$

$$\Rightarrow h(k) = -\left(h^*(M-1-k)\right)$$

Usually, one is interested in filters with **real-valued** coefficients, or see [link] and [link].



$\theta_0 = 0$ (Symmetric Filters).
$h(k) = h(M-1-k)$.



$\theta_0 = \frac{\pi}{2}$ (Anti-Symmetric Filters).
$h(k) = -h(M-1-k)$.

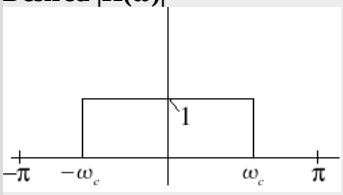
Filter design techniques are usually slightly different for each of these four different filter types. We will study the most common case, symmetric-odd length, in detail, and often leave the others for homework or tests or for when one encounters them in practice. Even-symmetric filters are often used; the anti-symmetric filters are rarely used in practice, except for special classes of filters, like differentiators or Hilbert transformers, in which the desired response is anti-symmetric.

So far, we have satisfied the condition that $H(\omega) = A(\omega)e^{-(i\theta_0)}e^{-\left(i\omega\frac{M-1}{2}\right)}$ where $A(\omega)$ is **real-valued**. However, we have **not** assured that $A(\omega)$ is **non-negative**. In general, this makes the design techniques much more difficult, so most FIR filter design methods actually design filters with **Generalized Linear Phase**:
$H(\omega) = A(\omega)e^{-\left(i\omega\frac{M-1}{2}\right)}$, where $A(\omega)$ is **real-valued**, but possible negative. $A(\omega)$ is called the **amplitude of the frequency response**.

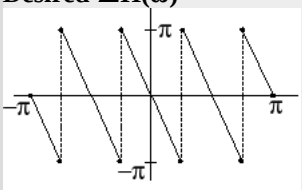
Note: $A(\omega)$ usually goes negative only in the stopband, and the stopband phase response is generally unimportant.

Note: $|H(\omega)| = \pm(A(\omega)) = A(\omega)e^{-\left(i\pi\frac{1}{2}(1-\text{sign}(A(\omega)))\right)}$ where $\text{sign}\ (x) = \begin{cases} 1 \text{ if } x > 0 \\ -1 \text{ if } x < 0 \end{cases}$

Example:
Lowpass Filter
Desired |H(ω)|

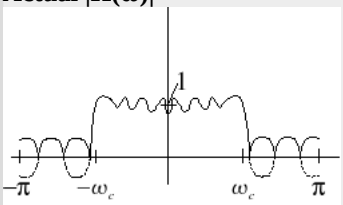


Desired ∠H(ω)



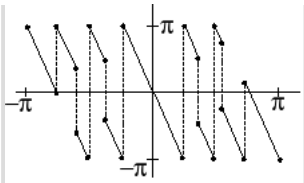
The slope of each line is $-\frac{M-1}{2}$.

Actual |H(ω)|



$A(\omega)$ goes negative.

Actual ∠H(ω)



2π phase jumps due to periodicity of phase. π phase jumps due to sign change in $A(\omega)$.

Time-delay introduces generalized linear phase.

Note: For odd-length FIR filters, a linear-phase design procedure is equivalent to a zero-phase design procedure followed by an $\frac{M-1}{2}$-sample delay of the impulse response. For even-length filters, the delay is non-integer, and the linear phase must be incorporated directly in the desired response!

Window Design Method

The truncate-and-delay design procedure is the simplest and most obvious FIR design procedure.
Exercise:

> **Problem:** Is it any Good?

> **Solution:**
>
> Yes; in fact it's optimal! (in a certain sense)

L2 optimization criterion

find $\forall n, 0 \leq n \leq M-1 : (h[n])$, maximizing the energy difference between the desired response and the actual response: i.e., find

$$\min_{h[n]} \left\{ h[n], \int_{-\pi}^{\pi} \left(|H_d(\omega) - H(\omega)| \right)^2 \, \mathrm{d}\, \omega \right\}$$

by Parseval's relationship
Equation:

$$\begin{aligned} \min_{h[n]} \left\{ h[n], \int_{-\pi}^{\pi} \left(|H_d(\omega) - H(\omega)| \right)^2 \, \mathrm{d}\, \omega \right\} &= 2\pi \sum_{n=-\infty}^{\infty} \left(|h_d[n] - h[n]| \right)^2 \\ &= 2\pi \left(\sum_{n=-\infty}^{-1} \left(|h_d[n] - h[n]| \right)^2 + \sum_{n=0}^{M-1} \left(|h_d[n] - h[n]| \right)^2 + \right. \end{aligned}$$

Since Math input error this becomes

$$\min_{h[n]} \left\{ h[n], \int_{-\pi}^{\pi} \left(|H_d(\omega) - H(\omega)| \right)^2 \, \mathrm{d}\, \omega \right\} = \sum_{h=-\infty}^{-1} \left(|h_d[n]| \right)^2 + \sum_{n=0}^{M-1} \left(|h[n] - h_d[n]| \right)^2 + \sum_{n=M}^{\infty} \left(|h_d[n]| \right)^2$$

Note: $h[n]$ has no influence on the first and last sums.

The best we can do is let

$$h[n] = \begin{cases} h_d[n] \ \text{ if } \ 0 \leq n \leq M-1 \\ 0 \ \text{ if } \ \text{else} \end{cases}$$

Thus $h[n] = h_d[n] w[n]$,

$$w[n] = \begin{cases} 1 \ \text{ if } \ 0 \leq n(M-1) \\ 0 \ \text{ if } \ \text{else} \end{cases}$$

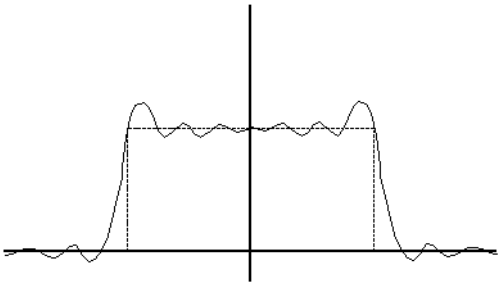
is **optimal** in a least-total-sqaured-error (L_2, or energy) sense!
Exercise:

> **Problem:** Why, then, is this design often considered undersirable?

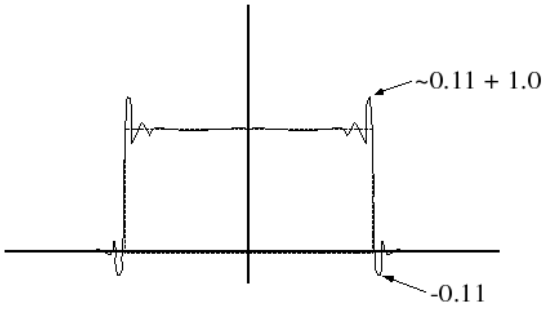
> **Solution:**

Gibbs Phenomenon

$A(\omega)$, small M



$A(\omega)$, large M



For desired spectra with discontinuities, the least-square designs are poor in a minimax (worst-case, or L_∞) error sense.

Window Design Method

Apply a more gradual truncation to reduce "ringing" (Gibb's Phenomenon)

$$\forall n 0 \leq n \leq M-1 h[n] = h_d[n] w[n] : (n 0 \leq n \leq M-1 h[n] = h_d[n] w[n])$$

Note: $H(\omega) = H_d(\omega) * W(\omega)$

The window design procedure (except for the boxcar window) is ad-hoc and not optimal in any usual sense. However, it is very simple, so it is sometimes used for "quick-and-dirty" designs of if the error criterion is itself heurisitic.

Frequency Sampling Design Method for FIR filters

Given a desired frequency response, the frequency sampling design method designs a filter with a frequency response **exactly** equal to the desired response at a particular set of frequencies ω_k.

Equation:

Procedure

$$\forall k, k = [\mathrm{o}, 1, \ldots, N-1] : \left(H_d(\omega_k) = \sum_{n=0}^{M-1} h(n) e^{-(i\omega_k n)} \right)$$

Note: Desired Response must incluce linear phase shift (if linear phase is desired)

Exercise:

Problem: What is $H_d(\omega)$ for an ideal lowpass filter, cotoff at ω_c?

Solution:

$$\begin{cases} e^{-(i\omega \frac{M-1}{2})} \text{ if } -\omega_c \leq \omega \leq \omega_c \\ 0 \text{ if } (-\pi \leq \omega < -\omega_c) \vee (\omega_c < \omega \leq \pi) \end{cases}$$

Note: This set of linear equations can be written in matrix form

Equation:

$$H_d(\omega_k) = \sum_{n=0}^{M-1} h(n) e^{-(i\omega_k n)}$$

Equation:

$$\begin{pmatrix} H_d(\omega_0) \\ H_d(\omega_1) \\ \vdots \\ H_d(\omega_{N-1}) \end{pmatrix} = \begin{pmatrix} e^{-(i\omega_0 0)} & e^{-(i\omega_0 1)} & \dots & e^{-(i\omega_0 (M-1))} \\ e^{-(i\omega_1 0)} & e^{-(i\omega_1 1)} & \dots & e^{-(i\omega_1 (M-1))} \\ \vdots & \vdots & \vdots & \vdots \\ e^{-(i\omega_{M-1} 0)} & e^{-(i\omega_{M-1} 1)} & \dots & e^{-(i\omega_{M-1} (M-1))} \end{pmatrix} \begin{pmatrix} h(0) \\ h(1) \\ \vdots \\ h(M-1) \end{pmatrix}$$

or

$$H_d = W\boldsymbol{h}$$

So
Equation:

$$\boldsymbol{h} = W^{-1} H_d$$

Note: W is a square matrix for $N = M$, and invertible as long as $\omega_i \neq \omega_j + 2\pi l$, $i \neq j$

Important Special Case

What if the frequencies are equally spaced between 0 and 2π, i.e. $\omega_k = \frac{2\pi k}{M} + \alpha$

Then

$$H_d(\omega_k) = \sum_{n=0}^{M-1} h(n) e^{-\left(i\frac{2\pi k n}{M}\right)} e^{-(i\alpha n)} = \sum_{n=0}^{M-1} \left(h(n) e^{-(i\alpha n)}\right) e^{-\left(i\frac{2\pi k n}{M}\right)} = \text{DFT!}$$

so

$$h(n) e^{-(i\alpha n)} = \frac{1}{M} \sum_{k=0}^{M-1} H_d(\omega_k) e^{i\frac{2\pi n k}{M}}$$

or

$$h[n] = \frac{e^{i\alpha n}}{M} \sum_{k=0}^{M-1} H_d[\omega_k] e^{i\frac{2\pi n k}{M}} = e^{i\alpha n}\, \text{IDFT}\left[H_d[\omega_k]\right]$$

Important Special Case #2

$h[n]$ symmetric, linear phase, and has real coefficients. Since $h[n] = h[M-n]$, there are only $\frac{M}{2}$ degrees of freedom, and only $\frac{M}{2}$ linear equations are required.
Equation:

$$\begin{aligned} H[\omega_k] &= \sum_{n=0}^{M-1} h[n] e^{-(i\omega_k n)} \\ &= \begin{cases} \sum_{n=0}^{\frac{M}{2}-1} h[n] \left(e^{-(i\omega_k n)} + e^{-(i\omega_k (M-n-1))} \right) & \text{if M even} \\ \sum_{n=0}^{M-\frac{3}{2}} h[n] \left(e^{-(i\omega_k n)} + e^{-(i\omega_k (M-n-1))} \right) \left(h\left[\frac{M-1}{2}\right] e^{-(i\omega_k \frac{M-1}{2})} \right) & \text{if M odd} \end{cases} \\ &= \begin{cases} e^{-(i\omega_k \frac{M-1}{2})} 2 \sum_{n=0}^{\frac{M}{2}-1} h[n] \cos\left(\omega_k \left(\frac{M-1}{2} - n\right)\right) & \text{if M even} \\ e^{-(i\omega_k \frac{M-1}{2})} 2 \sum_{n=0}^{M-\frac{3}{2}} h[n] \cos\left(\omega_k \left(\frac{M-1}{2} - n\right)\right) + h\left[\frac{M-1}{2}\right] & \text{if M odd} \end{cases} \end{aligned}$$

Removing linear phase from both sides yields

$$A(\omega_k) = \begin{cases} 2 \sum_{n=0}^{\frac{M}{2}-1} h[n] \cos\left(\omega_k \left(\frac{M-1}{2} - n\right)\right) & \text{if M even} \\ 2 \sum_{n=0}^{M-\frac{3}{2}} h[n] \cos\left(\omega_k \left(\frac{M-1}{2} - n\right)\right) + h\left[\frac{M-1}{2}\right] & \text{if M odd} \end{cases}$$

Due to symmetry of response for real coefficients, only $\frac{M}{2}$ ω_k on $\omega \in [0, \pi)$ need be specified, with the frequencies $-\omega_k$ thereby being implicitly defined also. Thus we have $\frac{M}{2}$ **real-valued** simultaneous linear equations to solve for $h[n]$.

Special Case 2a

$h[n]$ symmetric, odd length, linear phase, real coefficients, and ω_k equally spaced:
$\forall k, 0 \leq k \leq M-1 : \left(\omega_k = \frac{n\pi k}{M}\right)$
Equation:

$$\begin{aligned} h[n] &= \text{IDFT}\left[H_d(\omega_k)\right] \\ &= \frac{1}{M} \sum_{k=0}^{M-1} A(\omega_k) e^{-(i\frac{2\pi k}{M})} \frac{M-1}{2} e^{i\frac{2\pi n k}{M}} \\ &= \frac{1}{M} \sum_{k=0}^{M-1} A(k) e^{i(\frac{2\pi k}{M}(n - \frac{M-1}{2}))} \end{aligned}$$

To yield real coefficients, $A(\omega)$ mus be symmetric

$$(A(\omega) = A(-\omega)) \Rightarrow (A[k] = A[M-k])$$

Equation:

$$\begin{aligned} h[n] &= \frac{1}{M}\left(A(0) + \sum_{k=1}^{\frac{M-1}{2}} A[k]\left(e^{i\frac{2\pi k}{M}\left(n-\frac{M-1}{2}\right)} + e^{-\left(i2\pi k\left(n-\frac{M-1}{2}\right)\right)}\right)\right) \\ &= \frac{1}{M}\left(A(0) + 2\sum_{k=1}^{\frac{M-1}{2}} A[k]\cos\left(\frac{2\pi k}{M}\left(n-\frac{M-1}{2}\right)\right)\right) \\ &= \frac{1}{M}\left(A(0) + 2\sum_{k=1}^{\frac{M-1}{2}} A[k](-1)^k\cos\left(\frac{2\pi k}{M}\left(n+\frac{1}{2}\right)\right)\right) \end{aligned}$$

Simlar equations exist for even lengths, anti-symmetric, and $\alpha = \frac{1}{2}$ filter forms.

Comments on frequency-sampled design

This method is simple conceptually and very efficient for equally spaced samples, since $h[n]$ can be computed using the IDFT.

$H(\omega)$ for a frequency sampled design goes **exactly** through the sample points, but it may be very far off from the desired response for $\omega \neq \omega_k$. This is the main problem with frequency sampled design.

Possible solution to this problem: specify more frequency samples than degrees of freedom, and minimize the total error in the frequency response at all of these samples.

Extended frequency sample design

For the samples $H(\omega_k)$ where $0 \leq k \leq M-1$ and $N > M$, find $h[n]$, where $0 \leq n \leq M-1$ minimizing $\parallel H_d(\omega_k) - H(\omega_k) \parallel$

For $\parallel l \parallel_\infty$ norm, this becomes a linear programming problem (standard packages availble!)

Here we will consider the $\parallel l \parallel_2$ norm.

To minimize the $\parallel l \parallel_2$ norm; that is, $\sum_{n=0}^{N-1} |H_d(\omega_k) - H(\omega_k)|$, we have an overdetermined set of linear equations:

$$\begin{pmatrix} e^{-(i\omega_0 0)} & \dots & e^{-(i\omega_0(M-1))} \\ \vdots & \vdots & \vdots \\ e^{-(i\omega_{N-1}0)} & \dots & e^{-(i\omega_{N-1}(M-1))} \end{pmatrix} \boldsymbol{h} = \begin{pmatrix} H_d(\omega_0) \\ H_d(\omega_1) \\ \vdots \\ H_d(\omega_{N-1}) \end{pmatrix}$$

or

$$W\boldsymbol{h} = H_d$$

The minimum error norm solution is well known to be $\boldsymbol{h} = \left(\overline{W}W\right)^{-1}\overline{W}H_d$; $\left(\overline{W}W\right)^{-1}\overline{W}$ is well known as the pseudo-inverse matrix.

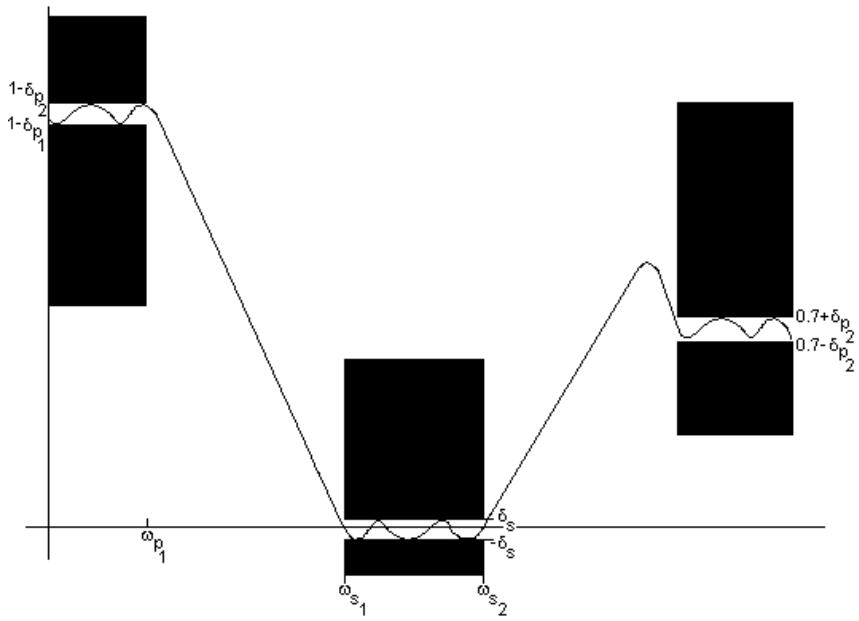
Note: Extended frequency sampled design discourages radical behavior of the frequency response between samples for sufficiently closely spaced samples. However, the actual frequency response may no longer pass exactly through **any** of the $H_d(\omega_k)$.

Parks-McClellan FIR Filter Design

The approximation tolerances for a filter are very often given in terms of the maximum, or worst-case, deviation within frequency bands. For example, we might wish a lowpass filter in a (16-bit) CD player to have no more than $\frac{1}{2}$-bit deviation in the pass and stop bands.

$$H(\omega) = \begin{cases} 1 - \frac{1}{2^{17}} \leq |H(\omega)| \leq 1 + \frac{1}{2^{17}} & \text{if} \quad |\omega| \leq \omega_p \\ \frac{1}{2^{17}} \geq |H(\omega)| & \text{if} \quad \omega_s \leq |\omega| \leq \pi \end{cases}$$

The Parks-McClellan filter design method efficiently designs linear-phase FIR filters that are optimal in terms of worst-case (minimax) error. Typically, we would like to have the shortest-length filter achieving these specifications. Figure [link] illustrates the amplitude frequency response of such a filter.



The black boxes on the left and right are the passbands, the black boxes in the middle represent the stop band, and the space between the boxes are the transition bands. Note that overshoots may be allowed in the transition bands.

Exercise:

Problem: Must there be a transition band?

Solution:

Yes, when the desired response is discontinuous. Since the frequency response of a finite-length filter must be continuous, without a transition band the worst-case error could be no less than half the discontinuity.

Formal Statement of the L-∞ (Minimax) Design Problem

For a given filter length (M) and type (odd length, symmetric, linear phase, for example), and a relative error weighting function $W(\omega)$, find the filter coefficients minimizing the maximum error

$$\underset{\boldsymbol{h}}{\operatorname{argmin}} \underset{\omega \in F}{\operatorname{argmax}} \, |E(\omega)| = \underset{\boldsymbol{h}}{\operatorname{argmin}} \, \| \, E(\omega) \, \|_\infty$$

where

$$E(\omega) = W(\omega) \left(H_d(\omega) - H(\omega) \right)$$

and F is a compact subset of $\omega \in [0, \pi]$ (i.e., all ω in the passbands and stop bands).

Note: Typically, we would often rather specify $\| \, E(\omega) \, \|_\infty \leq \delta$ and minimize over M and $\boldsymbol{h}$; however, the design techniques minimize δ for a given M. One then repeats the design procedure for different M until the minimum M satisfying the requirements is found.

We will discuss in detail the design only of odd-length symmetric linear-phase FIR filters. Even-length and anti-symmetric linear phase FIR filters are essentially the same except for a slightly different implicit weighting function. For arbitrary phase, exactly optimal design procedures have only recently been developed (1990).

Outline of L-∞ Filter Design

The Parks-McClellan method adopts an indirect method for finding the minimax-optimal filter coefficients.

1. Using results from Approximation Theory, simple conditions for determining whether a given filter is L^∞ (minimax) optimal are found.
2. An iterative method for finding a filter which satisfies these conditions (and which is thus optimal) is developed.

That is, the L^∞ filter design problem is actually solved **indirectly**.

Conditions for L-∞ Optimality of a Linear-phase FIR Filter

All conditions are based on Chebyshev's "Alternation Theorem," a mathematical fact from polynomial approximation theory.

Alternation Theorem

Let F be a compact subset on the real axis x, and let $P(x)$ be and Lth-order polynomial

$$P(x) = \sum_{k=0}^{L} a_k x^k$$

Also, let $D(x)$ be a desired function of x that is continuous on F, and $W(x)$ a positive, continuous weighting function on F. Define the error $E(x)$ on F as

$$E(x) = W(x)\left(D(x) - P(x)\right)$$

and

$$\parallel E(x) \parallel_\infty = \operatorname*{argmax}_{x \in F} |E(x)|$$

A necessary and sufficient condition that $P(x)$ is the unique Lth-order polynomial minimizing $\parallel E(x) \parallel_\infty$ is that $E(x)$ exhibits **at least** $L+2$ "alternations;" that is, there must exist at least $L+2$ values of x, $x_k \in F$, $k = [0, 1, \ldots, L+1]$, such that $x_0 < x_1 < \ldots < x_{L+2}$ and such that $E(x_k) = -E(x_{k+1}) = \pm(\parallel E \parallel_\infty)$
Exercise:

Problem: What does this have to do with linear-phase filter design?

Solution:

It's the same problem! To show that, consider an odd-length, symmetric linear phase filter.
Equation:

$$\begin{aligned} H(\omega) &= \textstyle\sum_{n=0}^{M-1} h(n) e^{-(i\omega n)} \\ &= e^{-\left(i\omega\frac{M-1}{2}\right)} \left(h\left(\tfrac{M-1}{2}\right) + 2\textstyle\sum_{n=1}^{L} h\left(\tfrac{M-1}{2} - n\right)\cos(\omega n)\right) \end{aligned}$$

Equation:

$$A(\omega) = h(L) + 2\sum_{n=1}^{L} h(L-n)\cos(\omega n)$$

Where $L \doteq \frac{M-1}{2}$.

Using trigonometric identities (such as
$\cos(n\alpha) = 2\cos((n-1)\alpha)\cos(\alpha) - \cos((n-2)\alpha)$), we can rewrite $A(\omega)$ as

$$A(\omega) = h(L) + 2\sum_{n=1}^{L} h(L-n)\cos(\omega n) = \sum_{k=0}^{L} \alpha_k \cos^k(\omega)$$

where the α_k are related to the $h(n)$ by a linear transformation. Now, let $x = \cos(\omega)$. This is a one-to-one mapping from $x \in [-1, 1]$ onto $\omega \in [0, \pi]$. Thus $A(\omega)$ is an Lth-order polynomial in $x = \cos(\omega)$!

Note: The alternation theorem holds for the L^∞ filter design problem, too!

Therefore, to determine whether or not a length-M, odd-length, symmetric linear-phase filter is optimal in an L^∞ sense, simply count the alternations in $E(\omega) = W(\omega)\left(A_d(\omega) - A(\omega)\right)$ in the pass and stop bands. If there are $L + 2 = \frac{M+3}{2}$ or more alternations, $h(n)$, $0 \leq n \leq M-1$ is the optimal filter!

Optimality Conditions for Even-length Symmetric Linear-phase Filters

For M even,

$$A(\omega) = \sum_{n=0}^{L} h(L-n)\cos\left(\omega\left(n + \frac{1}{2}\right)\right)$$

where $L = \frac{M}{2} - 1$ Using the trigonometric identity $\cos(\alpha+\beta) = \cos(\alpha-\beta) + 2\cos(\alpha)\cos(\beta)$ to pull out the $\frac{\omega}{2}$ term and then using the other trig identities, it can be shown that $A(\omega)$ can be written as

$$A(\omega) = \cos\left(\frac{\omega}{2}\right)\sum_{k=0}^{L} \alpha_k \cos^k(\omega)$$

Again, this is a polynomial in $x = \cos(\omega)$, except for a weighting function out in front.
Equation:

$$\begin{aligned} E(\omega) &= W(\omega)\left(A_d(\omega) - A(\omega)\right) \\ &= W(\omega)\left(A_d(\omega) - \cos\left(\tfrac{\omega}{2}\right)P(\omega)\right) \\ &= W(\omega)\cos\left(\tfrac{\omega}{2}\right)\left(\frac{A_d(\omega)}{\cos\left(\frac{\omega}{2}\right)} - P(\omega)\right) \end{aligned}$$

which implies
Equation:

$$E(x) = W'(x)\left(A'_d(x) - P(x)\right)$$

where

$$W'(x) = W\Big((\cos(x))^{-1}\Big)\cos\left(\frac{1}{2}(\cos(x))^{-1}\right)$$

and

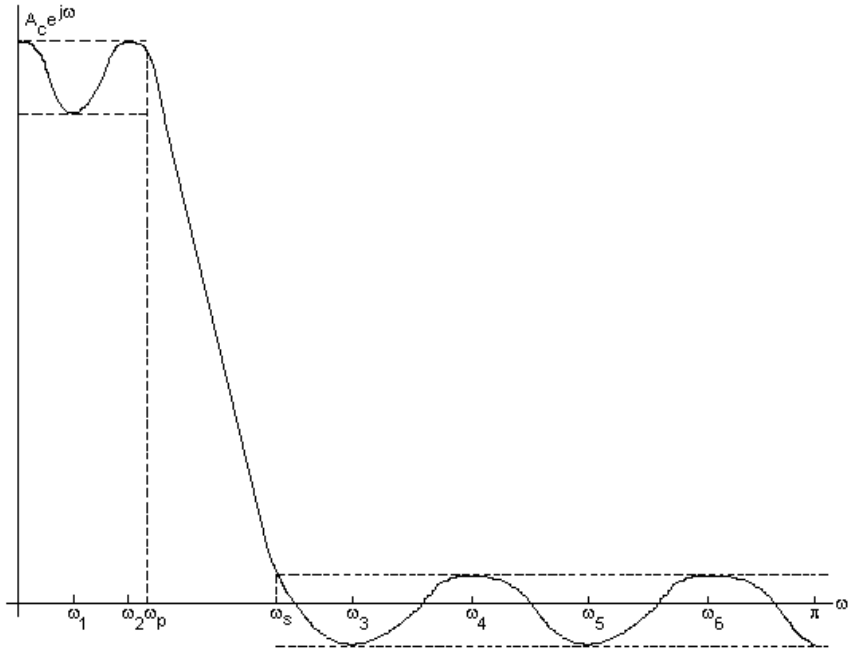
$$A'_d(x) = \frac{A_d\Big((\cos(x))^{-1}\Big)}{\cos\left(\frac{1}{2}(\cos(x))^{-1}\right)}$$

Again, this is a polynomial approximation problem, so the alternation theorem holds. If $E(\omega)$ has at least $L + 2 = \frac{M}{2} + 1$ alternations, the even-length symmetric filter is optimal in an L^∞ sense.

The prototypical filter design problem:

$$W = \begin{cases} 1 \text{ if } & |\omega| \leq \omega_p \\ \frac{\delta_s}{\delta_p} \text{ if } & |\omega_s| \leq |\omega| \end{cases}$$

See [link].



L-∞ Optimal Lowpass Filter Design Lemma

1. The maximum possible number of alternations for a lowpass filter is $L+3$: The proof is that the extrema of a polynomial occur only where the derivative is zero: $\frac{\partial P(x)}{\partial x} = 0$. Since $P'(x)$ is an $(L-1)$th-order polynomial, it can have at **most** $L-1$ zeros. **However**, the mapping $x = \cos(\omega)$ implies that $\frac{\partial A(\omega)}{\partial \omega} = 0$ at $\omega = 0$ and $\omega = \pi$, for two more possible alternation points. **Finally**, the band edges can also be alternations, for a total of $L-1+2+2 = L+3$ possible alternations.
2. There must be an alternation at either $\omega = 0$ or $\omega = \pi$.
3. Alternations must occur at ω_p and ω_s. See [link].
4. The filter must be equiripple except at possibly $\omega = 0$ or $\omega = \pi$. Again see [link].

Note: The alternation theorem doesn't directly suggest a method for computing the optimal filter. It simply tells us how to recognize that a filter **is** optimal, or **isn't** optimal. What we need is an intelligent way of guessing the optimal filter coefficients.

In matrix form, these $L+2$ simultaneous equations become

$$\begin{pmatrix} 1 & \cos(\omega_0) & \cos(2\omega_0) & \dots & \cos(L\omega_0) & \frac{1}{W(\omega_0)} \\ 1 & \cos(\omega_1) & \cos(2\omega_1) & \dots & \cos(L\omega_1) & \frac{-1}{W(\omega_1)} \\ \vdots & \vdots & \ddots & \dots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \dots & \ddots & \vdots \\ 1 & \cos(\omega_{L+1}) & \cos(2\omega_{L+1}) & \dots & \cos(L\omega_{L+1}) & \frac{\pm(1)}{W(\omega_{L+1})} \end{pmatrix} \begin{pmatrix} h(L) \\ h(L-1) \\ \vdots \\ h(1) \\ h(0) \\ \delta \end{pmatrix} = \begin{pmatrix} A_d(\omega_0) \\ A_d(\omega_1) \\ \vdots \\ \vdots \\ \vdots \\ A_d(\omega_{L+1}) \end{pmatrix}$$

or

$$W \begin{pmatrix} \boldsymbol{h} \\ \delta \end{pmatrix} = A_d$$

So, for the given set of $L+2$ extremal frequencies, we can solve for $\boldsymbol{h}$ and δ via $(\boldsymbol{h}\delta)^T = W^{-1}A_d$. Using the FFT, we can compute $A(\omega)$ of $h(n)$, on a dense set of frequencies. If the old ω_k are, in fact the extremal locations of $A(\omega)$, then the alternation theorem is satisfied and $h(n)$ is **optimal**. If not, repeat the process with the new extremal locations.

Computational Cost

$O\left(L^3\right)$ for the matrix inverse and $N \log_2 N$ for the FFT ($N \geq 32L$, typically), **per iteration**!

This method is expensive computationally due to the matrix inverse.

A more efficient variation of this method was developed by Parks and McClellan (1972), and is based on the Remez exchange algorithm. To understand the Remez exchange algorithm, we first need to understand Lagrange Interpoloation.

Now $A(\omega)$ is an Lth-order polynomial in $x = \cos(\omega)$, so Lagrange interpolation can be used to **exactly** compute $A(\omega)$ from $L+1$ samples of $A(\omega_k)$, $k = [0, 1, 2, ..., L]$.

Thus, given a set of extremal frequencies and knowing δ, samples of the amplitude response $A(\omega)$ can be computed **directly** from the
Equation:

$$A(\omega_k) = \frac{(-1)^{k(1)}}{W(\omega_k)}\delta + A_d(\omega_k)$$

without solving for the filter coefficients!

This leads to computational savings!

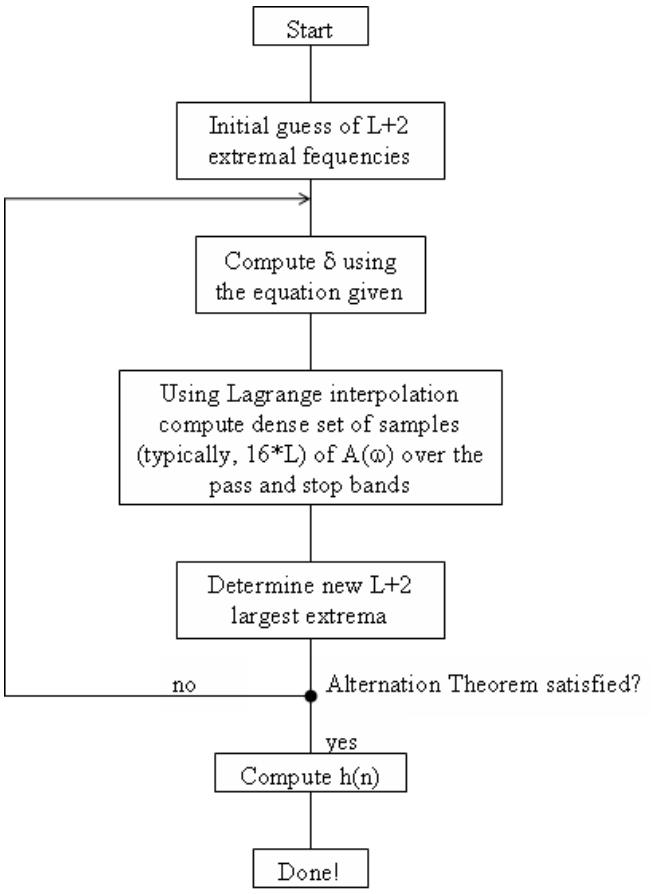
Note that [link] is a set of $L+2$ simultaneous equations, which can be solved for δ to obtain (Rabiner, 1975)
Equation:

$$\delta = \frac{\sum_{k=0}^{L+1} \gamma_k A_d(\omega_k)}{\sum_{k=0}^{L+1} \frac{(-1)^{k(1)}\gamma_k}{W(\omega_k)}}$$

where

$$\gamma_k = \prod_{i=i\neq k,0}^{L+1} \frac{1}{\cos(\omega_k) - \cos(\omega_i)}$$

The result is the Parks-McClellan FIR filter design method, which is simply an application of the Remez exchange algorithm to the filter design problem. See [link].


Start
Initial guess of L+2 extremal fequencies
Compute δ using the equation given
Using Lagrange interpolation compute dense set of samples (typically, 16*L) of A(ω) over the pass and stop bands
Determine new L+2 largest extrema
Alternation Theorem satisfied?
no
yes
Compute h(n)
Done!

The initial guess of extremal frequencies is usually equally spaced in the band. Computing δ costs $O\left(L^2\right)$. Using Lagrange interpolation costs $O(16LL) \simeq O\left(16L^2\right)$. Computing $h(n)$ costs $O\left(L^3\right)$, but it is only done once!

The cost per iteration is $O\left(16L^2\right)$, as opposed to $O\left(L^3\right)$; much more efficient for large L. Can also interpolate to DFT sample frequencies, take inverse FFT to get corresponding filter coefficients, and zeropad and take longer FFT to efficiently interpolate.

Lagrange Interpolation

Lagrange's interpolation method is a simple and clever way of finding the unique L th-order polynomial that exactly passes through $L+1$ distinct samples of a signal. Once the polynomial is known, its value can easily be interpolated at any point using the polynomial equation. Lagrange interpolation is useful in many applications, including Parks-McClellan FIR Filter Design.

Lagrange interpolation formula

Given an Lth-order polynomial

$$P(x) = a_0 + a_1 x + ... + a_L x^L = \sum_{k=0}^{L} a_k x^k$$

and $L+1$ values of $P(x_k)$ at different x_k, $k \in \{0, 1, ..., L\}$, $x_i \neq x_j$, $i \neq j$, the polynomial can be written as

$$P(x) = \sum_{k=0}^{L} P(x_k) \frac{(x - x_1)(x - x_2)...(x - x_{k-1})(x - x_{k+1})...(x - x_L)}{(x_k - x_1)(x_k - x_2)...(x_k - x_{k-1})(x_k - x_{k+1})...(x_k - x_L)}$$

The value of this polynomial at other x can be computed via substitution into this formula, or by expanding this formula to determine the polynomial coefficients a_k in standard form.

Proof

Note that for each term in the Lagrange interpolation formula above,

$$\prod_{i=0,,i\neq k}^{L} \frac{x - x_i}{x_k - x_i} = \begin{cases} 1 \text{ if } x = x_k \\ 0 \text{ if } (x = x_j) \wedge (j \neq k) \end{cases}$$

and that it is an Lth-order polynomial in x. The Lagrange interpolation formula is thus exactly equal to $P(x_k)$ at all x_k, and as a sum of Lth-order polynomials is itself an Lth-order polynomial.

It can be shown that the Vandermonde matrix

$$\begin{matrix} 1 & x_0 & {x_0}^2 & ... & {x_0}^L \\ 1 & x_1 & {x_1}^2 & ... & {x_1}^L \\ 1 & x_2 & {x_2}^2 & ... & {x_2}^L \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_L & {x_L}^2 & ... & {x_L}^L \end{matrix} \begin{matrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_L \end{matrix} = \begin{matrix} P(x_0) \\ P(x_1) \\ P(x_2) \\ \vdots \\ P(x_L) \end{matrix}$$

has a non-zero determinant and is thus invertible, so the Lth-order polynomial passing through all $L+1$ sample points x_j is unique. Thus the Lagrange polynomial expressions, as an Lth-order polynomial passing through the $L+1$ sample points, must be the unique $P(x)$.

Overview of IIR Filter Design

IIR Filter

$$y(n) = -\sum_{k=1}^{M-1} a_k y(n-k) + \sum_{k=0}^{M-1} b_k x(n-k)$$

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + ... + b_M z^{-M}}{1 + a_1 z^{-1} + a_2 z^{-2} + ... + a_M z^{-M}}$$

IIR Filter Design Problem

Choose $\{a_i\}$, $\{b_i\}$ to **best** approximate some desired $|H_d(w)|$ or, (occasionally), $H_d(w)$.

As before, different design techniques will be developed for different approximation criteria.

Outline of IIR Filter Design Material

- **Bilinear Transform** Maps $\| L \|_\infty$ optimal (and other) **analog** filter designs to $\| L \|_\infty$ optimal digital IIR filter designs.
- **Prony's Method** Quasi-$\| L \|_2$ optimal method for time-domain fitting of a desired impulse response (ad hoc).
- **Lp Optimal Design** $\| L \|_p$ optimal filter design ($1 < p < \infty$) using non-linear optimization techniques.

Comments on IIR Filter Design Methods

The bilinear transform method is used to design "typical" $\| L \|_\infty$ magnitude optimal filters. The $\| L \|_p$ optimization procedures are used to design filters for which classical analog prototype solutions don't exist. The program by Deczky (*DSP Programs Book*, IEEE Press) is widely used. Prony/Linear Prediction techniques are used often to obtain initial guesses,

and are almost exclusively used in data modeling, system identification, and most applications involving the fitting of real data (for example, the impulse response of an unknown filter).

Prototype Analog Filter Design

Analog Filter Design

Laplace transform:

$$H(s) = \int_{-\infty}^{\infty} h_a(t) e^{-(st)} \, \mathrm{d}\, t$$

Note that the continuous-time Fourier transform is $H(i\lambda)$ (the Laplace transform evaluated on the imaginary axis).

Since the early 1900's, there has been a lot of research on designing analog filters of the form

$$H(s) = \frac{b_0 + b_1 s + b_2 s^2 + ... + b_M s^M}{1 + a_1 s + a_2 s^2 + ... + a_M s^M}$$

A causal IIR filter **cannot** have linear phase (no possible symmetry point), and design work for analog filters has concentrated on designing filters with equiriplle ($\| L \|_\infty$) **magnitude** responses. These design problems have been solved. We will not concern ourselves here with the design of the analog prototype filters, only with how these designs are mapped to discrete-time while preserving optimality.

An analog filter with **real** coefficients must have a magnitude response of the form

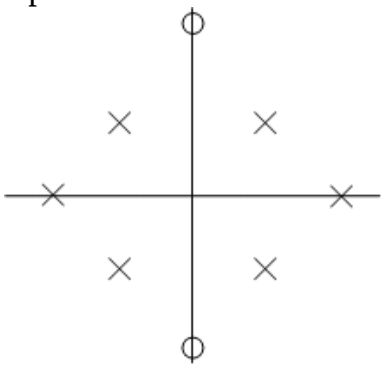
$$\left(|H(\lambda)|\right)^2 = B\left(\lambda^2\right)$$

Equation:

$$\begin{aligned}
H(i\lambda)H(i\lambda) &= \frac{b_0+b_1 i\lambda+b_2(i\lambda)^2+b_3(i\lambda)^3+...}{1+a_1 i\lambda+a_2(i\lambda)^2+...} H(i\lambda) \\
&= \frac{b_0-b_2\lambda^2+b_4\lambda^4+...+i\lambda\left(b_1-b_3\lambda^2+b_5\lambda^4+...\right)}{1-a_2\lambda^2+a_4\lambda^4+...+i\lambda(a_1-a_3\lambda^2+a_5\lambda^4+...)} \frac{b_0-b_2\lambda^2+b_4\lambda^4+...+i\lambda(b_1-b_3\lambda^2+b_5\lambda^4+...)}{1-a_2\lambda^2+a_4\lambda^4+...+i\lambda(a_1-a_3\lambda^2+a_5\lambda^4+...)} \\
&= \frac{\left(b_0-b_2\lambda^2+b_4\lambda^4+...\right)^2+\lambda^2\left(b_1-b_3\lambda^2+b_5\lambda^4+...\right)^2}{\left(1-a_2\lambda^2+a_4\lambda^4+...\right)^2+\lambda^2(a_1-a_3\lambda^2+a_5\lambda^4+...)^2} \\
&= B\left(\lambda^2\right)
\end{aligned}$$

Let $s = i\lambda$, note that the poles and zeros of $B\left(-s^2\right)$ are symmetric around **both** the real and imaginary axes: that is, a pole at p_1 implies poles at p_1, p_1, $-p_1$, and $-p_1$, as seen in [link].

s-plane



Recall that an analog filter is stable and causal if all the poles are in the left half-plane, LHP, and is **minimum phase** if all zeros and poles are in the LHP.

$s = i\lambda$: $B\left(\lambda^2\right) = B\left(-s^2\right) = H(s)H(-s) = H\left(i\lambda\right)H\left(-\left(i\lambda\right)\right) = H\left(i\lambda\right)H(i\lambda)$ we can factor $B\left(-s^2\right)$ into $H(s)H(-s)$, where $H(s)$ has the left half plane poles and zeros, and $H(-s)$ has the RHP poles and zeros.

$\left(|H(s)|\right)^2 = H(s)H(-s)$ for $s = i\lambda$, so $H(s)$ has the magnitude response $B\left(\lambda^2\right)$. The trick to analog filter design is to design a good $B\left(\lambda^2\right)$, then factor this to obtain a filter with that **magnitude** response.

The traditional analog filter designs all take the form $B\left(\lambda^2\right) = \left(|H(\lambda)|\right)^2 = \frac{1}{1+F(\lambda^2)}$, where F is a rational function in λ^2.

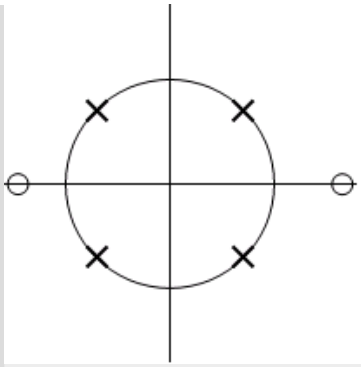
Example:

$$B\left(\lambda^2\right) = \frac{2+\lambda^2}{1+\lambda^4}$$

$$B\left(-s^2\right) = \frac{2-s^2}{1+s^4} = \frac{\left(\sqrt{2}-s\right)\left(\sqrt{2}+s\right)}{\left(s+\alpha\right)\left(s-\alpha\right)\left(s+\alpha\right)\left(s-\alpha\right)}$$

where $\alpha = \frac{1+i}{\sqrt{2}}$.

Note: Roots of $1+s^N$ are N points equally spaced around the unit circle ([link]).



Take $H(s) =$ LHP factors:

$$H(s) = \frac{\sqrt{2}+s}{(s+\alpha)(s+\alpha)} = \frac{\sqrt{2}+s}{s^2+\sqrt{2}s+1}$$

Traditional Filter Designs

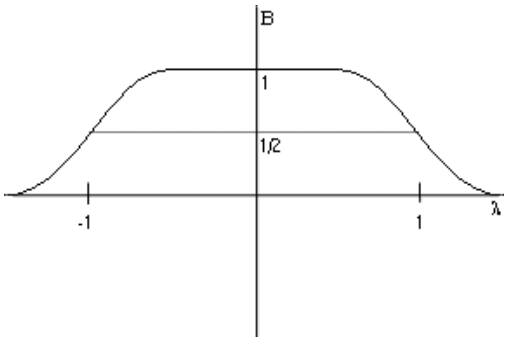
Butterworth

$$B(\lambda^2) = \frac{1}{1+\lambda^{2M}}$$

Note: Remember this for homework and rest problems!

"Maximally smooth" at $\lambda = 0$ and $\lambda = \infty$ (maximum possible number of zero derivatives). [link].

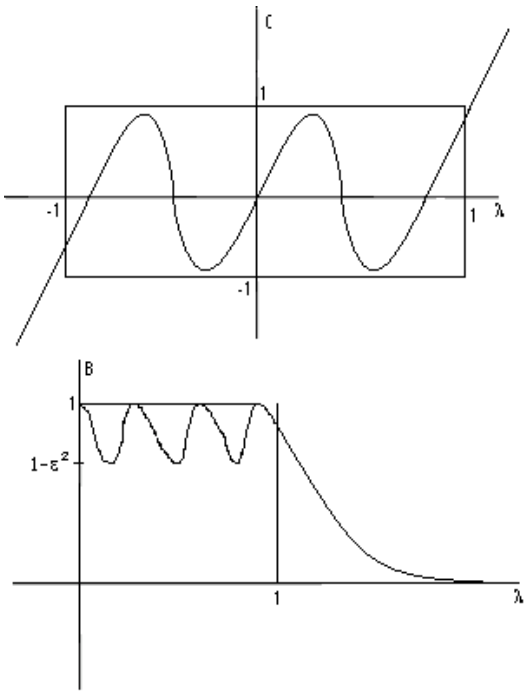
$$B(\lambda^2) = (|H(\lambda)|)^2$$



Chebyshev

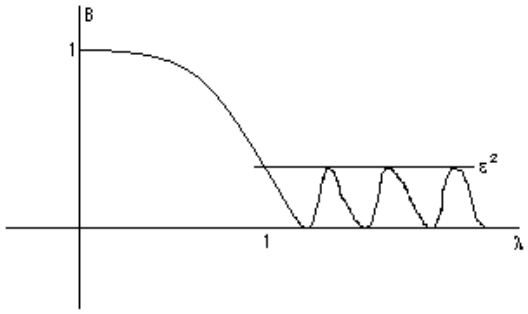
$$B(\lambda^2) = \frac{1}{1 + \varepsilon^2 {C_M}^2(\lambda)}$$

where ${C_M}^2(\lambda)$ is an Mth order Chebyshev polynomial. [link].



Inverse Chebyshev

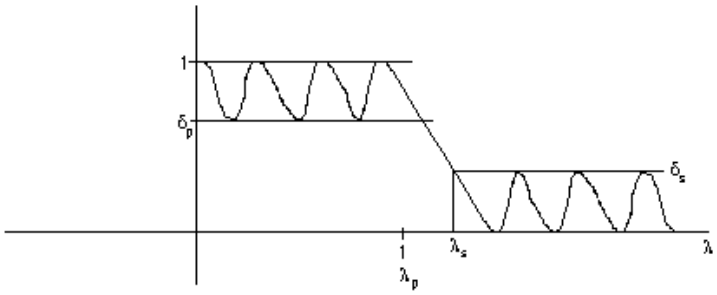
[link].



Elliptic Function Filter (Cauer Filter)

$$B(\lambda^2) = \frac{1}{1 + \varepsilon^2 {J_M}^2(\lambda)}$$

where J_M is the "Jacobi Elliptic Function." [link].



The Cauer filter is $\| \boldsymbol{L} \|_\infty$ optimum in the sense that for a given M, δ_p, δ_s, and λ_p, the transition bandwidth is smallest.

That is, it is $\| \boldsymbol{L} \|_\infty$ optimal.

IIR Digital Filter Design via the Bilinear Transform

A **bilinear transform** maps an analog filter $H_a(s)$ to a discrete-time filter $H(z)$ of the same order.

If only we could somehow map these optimal analog filter designs to the digital world while preserving the magnitude response characteristics, we could make use of the already-existing body of knowledge concerning optimal analog filter design.

Bilinear Transformation

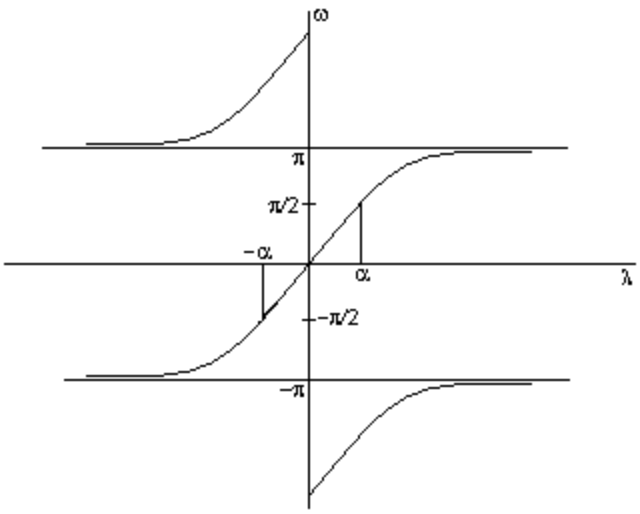
The Bilinear Transform is a nonlinear $\mathbb{C} \to \mathbb{C}$ mapping that maps a function of the complex variable s to a function of a complex variable z. This map has the property that the LHP in s ($\Re(s) < 0$) maps to the interior of the unit circle in z, and the $i\lambda = s$ axis maps to the unit circle $e^{i\omega}$ in z.

Bilinear transform:

$$s = \alpha \frac{z-1}{z+1}$$

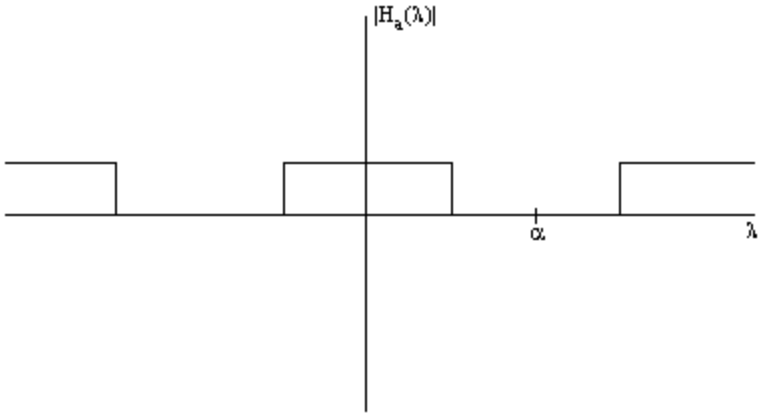
$$H(z) = H_a\left(s = \alpha \frac{z-1}{z+1}\right)$$

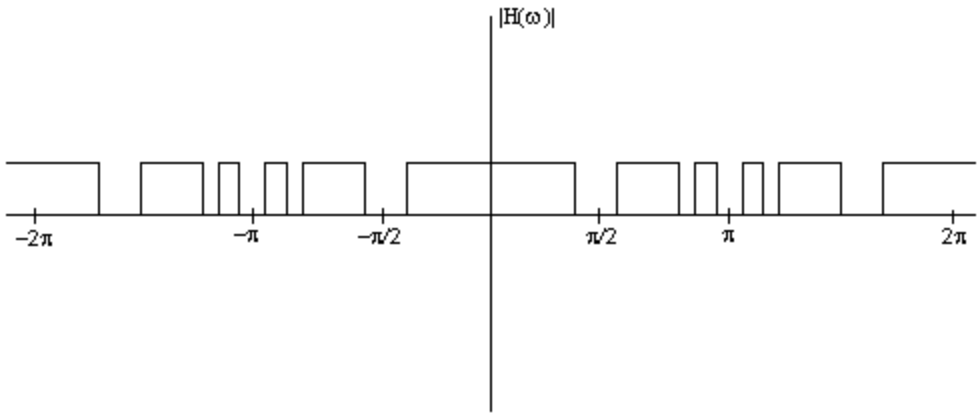
Note: $i\lambda = \alpha \frac{e^{i\omega}-1}{e^{i\omega}+1} = \alpha \frac{\left(e^{i\omega}-1\right)\left(e^{-(i\omega)}+1\right)}{\left(e^{i\omega}+1\right)\left(e^{-(i\omega)}+1\right)} = \frac{2i\sin(\omega)}{2+2\cos(\omega)} = i\alpha\tan\left(\frac{\omega}{2}\right)$, so $\lambda \equiv \alpha\tan\left(\frac{\omega}{2}\right), \omega \equiv 2\arctan\left(\frac{\lambda}{\alpha}\right)$. [link].



The magnitude response doesn't change in the mapping from λ to ω, it is simply warped nonlinearly according to $H(\omega) = H_a\left(\alpha \tan\left(\frac{\omega}{2}\right)\right)$, [link].

The first image implies the second one.





Note: This mapping preserves $\| L \|_\infty$ errors in (warped) frequency bands. Thus optimal Cauer ($\| L \|_\infty$) filters in the analog realm can be mapped to $\| L \|_\infty$ optimal discrete-time IIR filters using the bilinear transform! This is how IIR filters with $\| L \|_\infty$ optimal magnitude responses are designed.

Note: The parameter α provides one degree of freedom which can be used to map a single λ_0 to any desired ω_0:

$$\lambda_0 = \alpha \tan\left(\frac{\omega_0}{2}\right)$$

or

$$\alpha = \frac{\lambda_0}{\tan\left(\frac{\omega_0}{2}\right)}$$

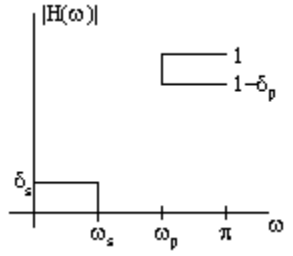
This can be used, for example, to map the pass-band edge of a lowpass analog prototype filter to any desired pass-band edge in ω. Often, analog prototype filters will be designed with $\lambda = 1$ as a band edge, and α will be used to locate the band edge in ω. Thus an Mth order optimal lowpass analog filter prototype can be used to design **any** M^{th} order discrete-time lowpass IIR filter with the same ripple specifications.

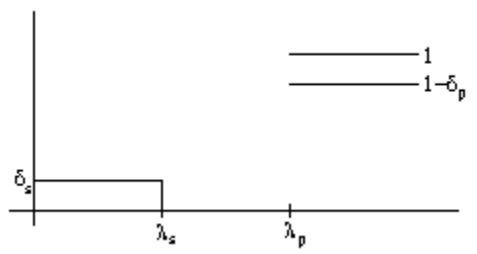
Prewarping

Given specifications on the frequency response of an IIR filter to be designed, map these to specifications in the analog frequency domain which are equivalent. Then a satisfactory analog prototype can be designed which, when transformed to discrete-time using the bilinear transformation, will meet the specifications.

Example:
The goal is to design a high-pass filter, $\omega_s = \omega_s$, $\omega_p = \omega_p$, $\delta_s = \delta_s$, $\delta_p = \delta_p$; pick up some $\alpha = \alpha_0$. In [link] the δ_i remain the same and the band edges are mapped by $\lambda_i = \alpha_0 \tan\left(\frac{\omega_i}{2}\right)$.

$$\text{Where } \lambda_s = \alpha_0 \tan\left(\frac{\omega_s}{2}\right) \text{ and } \lambda_p = \alpha_0 \tan\left(\frac{\omega_p}{2}\right).$$





Impulse-Invariant Design

Pre-classical, adhoc-but-easy method of converting an analog prototype filter to a digital IIR filter. Does not preserve any optimality.

Impulse invariance means that digital filter impulse response exactly equals samples of the analog prototype impulse response:

$$\forall n : (h(n) = h_a(nT))$$

How is this done?

The impulse response of a causal, stable analog filter is simply a sum of decaying exponentials:

$$H_a(s) = \frac{b_0 + b_1 s + b_2 s^2 + ... + b_p s^p}{1 + a_1 s + a_2 s^2 + ... + a_p s^p} = \frac{A_1}{s - s_1} + \frac{A_2}{s - s_2} + ... + \frac{A_p}{s - s_p}$$

which implies

$$h_a(t) = \left(A_1 e^{s_1 t} + A_2 e^{s_2 t} + ... + A_p e^{s_p t}\right) u(t)$$

For impulse invariance, we desire

$$h(n) = h_a(nT) = \left(A_1 e^{s_1 nT} + A_2 e^{s_2 nT} + ... + A_p e^{s_p nT}\right) u(n)$$

Since

$$A_k e^{(s_k T)n} u(n) \equiv \frac{A_k z}{z - e^{s_k T}}$$

where $|z| > \left|e^{s_k T}\right|$, and

$$H(z) = \sum_{k=1}^{p} A_k \frac{z}{z - e^{s_k T}}$$

where $|z| > \max_k \left\{k, \left|e^{s_k T}\right|\right\}$.

This technique is used occasionally in digital simulations of analog filters.

Exercise:

Problem:

What is the main problem/drawback with this design technique?

Solution:

Since it samples the non-bandlimited impulse response of the analog prototype filter, the frequency response **aliases**. This distorts the original analog frequency and destroys any optimal frequency properties in the resulting digital filter.

Digital-to-Digital Frequency Transformations

Given a prototype **digital** filter design, transformations similar to the bilinear transform can also be developed.

Requirements on such a mapping $z^{-1} = g\left(z^{-1}\right)$:

1. points inside the unit circle stay inside the unit circle (condition to preserve stability)
2. unit circle is mapped to itself (preserves frequency response)

This condition implies that $e^{-(i\omega_1)} = g\left(e^{-(i\omega)}\right) = |g(\omega)|e^{i\angle(g(\omega))}$ requires that $\left|g\left(e^{-(i\omega)}\right)\right| = 1$ on the unit circle!

Thus we require an **all-pass** transformation:

$$g\left(z^{-1}\right) = \prod_{k=1}^{p} \frac{z^{-1} - \alpha_k}{1 - \alpha_k z^{-1}}$$

where $|\alpha_K| < 1$, which is required to satisfy this condition.

Example:
Lowpass-to-Lowpass

$$z_1{}^{-1} = \frac{z^{-1} - a}{1 - az^{-1}}$$

which maps original filter with a cutoff at ω_c to a new filter with cutoff ω_c',

$$a = \frac{\sin\left(\frac{1}{2}\left(\omega_c - \omega_c'\right)\right)}{\sin\left(\frac{1}{2}\left(\omega_c + \omega_c'\right)\right)}$$

Example:
Lowpass-to-Highpass

$$z_1{}^{-1} = \frac{z^{-1} + a}{1 + az^{-1}}$$

which maps original filter with a cutoff at ω_c to a frequency reversed filter with cutoff ω_c',

$$a = \frac{\cos\left(\frac{1}{2}\left(\omega_c - \omega_c'\right)\right)}{\cos\left(\frac{1}{2}\left(\omega_c + \omega_c'\right)\right)}$$

(Interesting and occasionally useful!)

Prony's Method

Prony's Method is a quasi-least-squares time-domain IIR filter design method.

First, assume $H(z)$ is an "all-pole" system:
Equation:

$$H(z) = \frac{b_0}{1 + \sum_{k=1}^{M} a_k z^{-k}}$$

and

$$h(\mathrm{n}) = -\sum_{k=1}^{M} a_k h(n-k) + b_0 \delta(n)$$

where $h(n) = 0$, $n < 0$ for a causal system.

Note: For $h = 0$, $h(0) = b_0$.

Let's attempt to fit a desired impulse response (let it be **causal**, although one can extend this technique when it isn't) $h_d(n)$.

A true least-squares solution would attempt to minimize

$$\varepsilon^2 = \sum_{n=0}^{\infty} \left(|h_d(n) - h(n)| \right)^2$$

where $H(z)$ takes the form in [link]. This is a difficult non-linear optimization problem which is known to be plagued by local minima in the error surface. So instead of solving this difficult non-linear problem, we solve the **deterministic linear prediction** problem, which is related to, **but not the same as**, the true least-squares optimization.

The deterministic linear prediction problem is a **linear** least-squares optimization, which is easy to solve, but it minimizes the **prediction** error, not the $(|\text{desired} - \text{actual}|)^2$ response error.

Notice that for $n > 0$, with the all-pole filter
Equation:

$$h(n) = -\sum_{k=1}^{M} a_k h(n-k)$$

the right hand side of this equation is a **linear predictor** of $h(n)$ in terms of the M previous samples of $h(n)$.

For the desired reponse $h_d(n)$, one can choose the recursive filter coefficients a_k to minimize the squared prediction error

$${\varepsilon_p}^2 = \sum_{n=1}^{\infty} \left(\left| h_d(n) + \sum_{k=1}^{M} a_k h_d(n-k) \right| \right)^2$$

where, in practice, the ∞ is replaced by an N.

In matrix form, that's

$$\begin{pmatrix} h_d(0) & 0 & \dots & 0 \\ h_d(1) & h_d(0) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ h_d(N-1) & h_d(N-2) & \dots & h_d(N-M) \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_M \end{pmatrix} \simeq - \begin{pmatrix} h_d(1) \\ h_d(2) \\ \vdots \\ h_d(N) \end{pmatrix}$$

or

$$H_d \boldsymbol{a} \simeq -h_d$$

The optimal solution is

$$a_{\text{lp}} = -\left(\left({H_d}^{\text{H}} H_d \right)^{-1} {H_d}^{\text{H}} h_d \right)$$

Now suppose $H(z)$ is an Mth-order IIR (ARMA) system,

$$H(z) = \frac{\sum_{k=0}^{M} b_k z^{-k}}{1 + \sum_{k=1}^{M} a_k z^{-k}}$$

or
Equation:

$$\begin{aligned} h(n) &= -\sum_{k=1}^{M} a_k h(n-k) + \sum_{k=0}^{M} b_k \delta(n-k) \\ &= \begin{cases} -\sum_{k=1}^{M} a_k h(n-k) + b_n & \text{if } 0 \le n \le M \\ -\sum_{k=1}^{M} a_k h(n-k) & \text{if } n > M \end{cases} \end{aligned}$$

For $n > M$, this is just like the all-pole case, so we can solve for the best predictor coefficients as before:

$$\begin{pmatrix} h_d(M) & h_d(M-1) & \dots & h_d(1) \\ h_d(M+1) & h_d(M) & \dots & h_d(2) \\ \vdots & \vdots & \ddots & \vdots \\ h_d(N-1) & h_d(N-2) & \dots & h_d(N-M) \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_M \end{pmatrix} \simeq \begin{pmatrix} h_d(M+1) \\ h_d(M+2) \\ \vdots \\ h_d(N) \end{pmatrix}$$

or

$$\widehat{H}_d \boldsymbol{a} \simeq \widehat{h}_d$$

and

$$a_{\text{opt}} = \left(\left(\widehat{H}_d \right)^{\text{H}} H_d \right)^{-1} {H_d}^{\text{H}} \widehat{h}_d$$

Having determined the a's, we can use them in [link] to obtain the b_n's:

$$b_n = \sum_{k=1}^{M} a_k h_d(n-k)$$

where $h_d(n-k) = 0$ for $n - k < 0$.

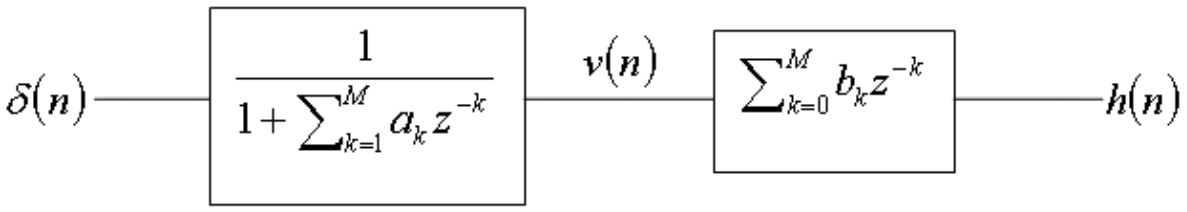
For $N = 2M$, $\widehat{H}_d$ is square, and we can solve **exactly** for the a_k's with no error. The b_k's are also chosen such that there is no error in the first $M + 1$ samples of $h(n)$. Thus for $N = 2M$, the first $2M + 1$ points of $h(n)$ **exactly equal** $h_d(n)$. This is called **Prony's Method**. Baron de Prony invented this in 1795.

For $N > 2M$, $h_d(n) = h(n)$ for $0 \leq n \leq M$, the prediction error is minimized for $M + 1 < n \leq N$, and whatever for $n \geq N + 1$. This is called the **Extended Prony Method**.

One might prefer a method which tries to minimize an overall error with the numerator coefficients, rather than just using them to exactly fit $h_d(0)$ to $h_d(M)$.

Shank's Method

1. Assume an all-pole model and fit $h_d(n)$ by minimizing the prediction error $1 \leq n \leq N$.
2. Compute $v(n)$, the impulse response of this all-pole filter.
3. Design an all-zero (MA, FIR) filter which fits $v(n)*h_z(n) \simeq h_d(n)$ optimally in a least-squares sense ([link]).



Here, $h(n) \simeq h_d(n)$.

The final IIR filter is the cascade of the all-pole and all-zero filter.

This is is solved by

$$\min_{b_k} \left\{ b_k, \sum_{n=0}^{N} \left(\left| h_d(n) - \sum_{k=0}^{M} b_k v(n-k) \right| \right)^2 \right\}$$

or in matrix form

$$\begin{pmatrix} v(0) & 0 & 0 & ... & 0 \\ v(1) & v(0) & 0 & ... & 0 \\ v(2) & v(1) & v(0) & ... & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ v(N) & v(N-1) & v(N-2) & ... & v(N-M) \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_M \end{pmatrix} \simeq \begin{pmatrix} h_d(0) \\ h_d(1) \\ h_d(2) \\ \vdots \\ h_d(N) \end{pmatrix}$$

Which has solution:

$$b_{\text{opt}} = \left(V^{\text{H}} V \right)^{-1} V^{\text{H}} \boldsymbol{h}$$

Notice that none of these methods solve the true least-squares problem:

$$\min_{\boldsymbol{a},\boldsymbol{b}} \left\{ \boldsymbol{a}, , , \boldsymbol{b}, \sum_{n=0}^{\infty} \left(|h_d(n) - h(n)| \right)^2 \right\}$$

which is a difficult non-linear optimization problem. The true least-squares problem can be written as:

$$\min_{\alpha,\beta} \left\{ \alpha, , , \beta, \sum_{n=0}^{\infty} \left(\left| h_d(n) - \sum_{i=1}^{M} \alpha_i e^{\beta_i n} \right| \right)^2 \right\}$$

since the impulse response of an IIR filter is a sum of exponentials, and non-linear optimization is then used to solve for the α_i and β_i.

Linear Prediction

Recall that for the all-pole design problem, we had the overdetermined set of linear equations:

$$\begin{pmatrix} h_d(0) & 0 & ... & 0 \\ h_d(1) & h_d(0) & ... & 0 \\ \vdots & \vdots & \ddots & \vdots \\ h_d(N-1) & h_d(N-2) & ... & h_d(N-M) \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_M \end{pmatrix} \simeq - \begin{pmatrix} h_d(1) \\ h_d(2) \\ \vdots \\ h_d(N) \end{pmatrix}$$

with solution $\boldsymbol{a} = \left({H_d}^{\mathrm{H}} H_d\right)^{-1} {H_d}^{\mathrm{H}} h_d$

Let's look more closely at ${H_d}^{\mathrm{H}} H_d = R$. r_{ij} is related to the **correlation** of h_d with itself:

$$r_{ij} = \sum_{k=0}^{N-\max\{i,j\}} h_d(k) h_d(k + |i-j|)$$

Note also that:

$${H_d}^{\mathrm{H}} h_d = \begin{pmatrix} r_d(1) \\ r_d(2) \\ r_d(3) \\ \vdots \\ r_d(M) \end{pmatrix}$$

where

$$r_d(i) = \sum_{n=0}^{N-i} h_d(n) h_d(n+i)$$

so this takes the form $a_{\text{opt}} = -\left(R^{\mathrm{H}} r_d\right)$, or $R\boldsymbol{a} = -\boldsymbol{r}$, where R is $M \times M$, $\boldsymbol{a}$ is $M \times 1$, and $\boldsymbol{r}$ is also $M \times 1$.

Except for the changing endpoints of the sum, $r_{ij} \simeq r(i-j) = r(j-i)$. If we tweak the problem slightly to make $r_{ij} = r(i-j)$, we get:

$$\begin{pmatrix} r(0) & r(1) & r(2) & ... & r(M-1) \\ r(1) & r(0) & r(1) & ... & \vdots \\ r(2) & r(1) & r(0) & ... & \vdots \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ r(M-1) & ... & ... & ... & r(0) \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_M \end{pmatrix} = - \begin{pmatrix} r(1) \\ r(2) \\ r(3) \\ \vdots \\ r(M) \end{pmatrix}$$

The matrix R is **Toeplitz** (diagonal elements equal), and $\boldsymbol{a}$ can be solved for with $O\left(M^2\right)$ computations using Levinson's recursion.

Statistical Linear Prediction

Used very often for forecasting (e.g. stock market).

Given a time-series $y(n)$, assumed to be produced by an auto-regressive (AR) (all-pole) system:

$$y(n) = -\sum_{k=1}^{M} a_k y(n-k) + u(n)$$

where $u(n)$ is a white Gaussian noise sequence which is stationary and has zero mean.

To determine the model parameters $\{a_k\}$ minimizing the variance of the prediction error, we seek
Equation:

$$\begin{aligned}\min_{a_k}\left\{a_k, E\left[\left(y(n)+\sum_{k=1}^{M} a_k y(n-k)\right)^2\right]\right\} &= \min_{a_k}\left\{a_k, E\left[y^2(n)+2\sum_{k=1}^{M} a_k y(n)y(n-k)+\sum_{k=1}^{M}\right.\right. \\ &= \min_{a_k}\left\{a_k, E\left[y^2(n)\right]+2\sum_{k=1}^{M} a_k E[y(n)y(n-k)]+\sum\right.\end{aligned}$$

Note: The mean of $y(n)$ is zero.

Equation:

$$\varepsilon^2 = r(0) + 2\begin{pmatrix} r(1) & r(2) & r(3) & \dots & r(M)\end{pmatrix}\begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_M\end{pmatrix} + \begin{pmatrix} a_1 & a_2 & a_3 & \dots & a_M\end{pmatrix}\begin{pmatrix} r(0) & r(1) & r(2) & \dots \\ r(1) & r(0) & r(1) & \dots \\ r(2) & r(1) & r(0) & \dots \\ \vdots & \vdots & \vdots & \ddots \\ r(M-1) & \dots & \dots & \dots\end{pmatrix}$$

Equation:

$$\frac{\partial \varepsilon^2}{\partial \boldsymbol{a}} = 2\boldsymbol{r} + 2R\boldsymbol{a}$$

Setting [link] equal to zero yields: $R\boldsymbol{a} = -\boldsymbol{r}$ These are called the **Yule-Walker** equations. In practice, given samples of a sequence $y(n)$, we estimate $r(n)$ as

$$\widehat{r(n)} = \frac{1}{N}\sum_{k=0}^{N-n} y(n)y(n+k) \simeq E[y(k)y(n+k)]$$

which is extremely similar to the deterministic least-squares technique.

Filter Structures

A realizable filter must require only a finite number of computations per output sample. For linear, causal, time-Invariant filters, this restricts one to rational transfer functions of the form

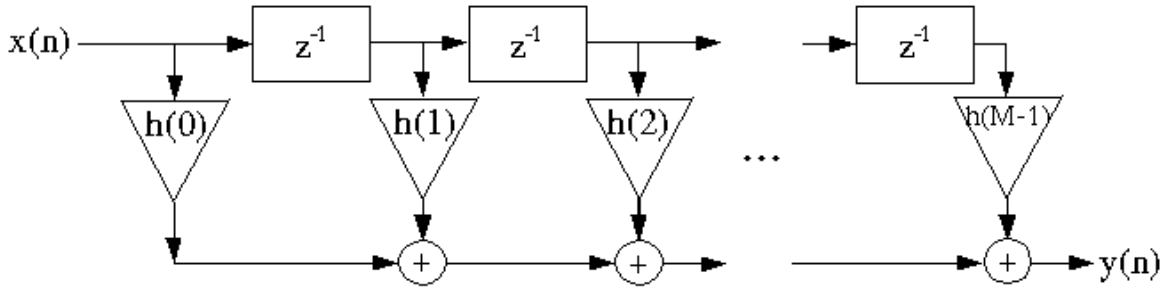
$$H(z) = \frac{b_0 + b_1 z^{-1} + \ldots + b_m z^{-m}}{1 + a_1 z^{-1} + a_2 z^{-2} + \ldots + a_n z^{-n}}$$

Assuming no pole-zero cancellations, $H(z)$ is FIR if $\forall i, i > 0 : (a_i = 0)$, and IIR otherwise. Filter structures usually implement rational transfer functions as difference equations.

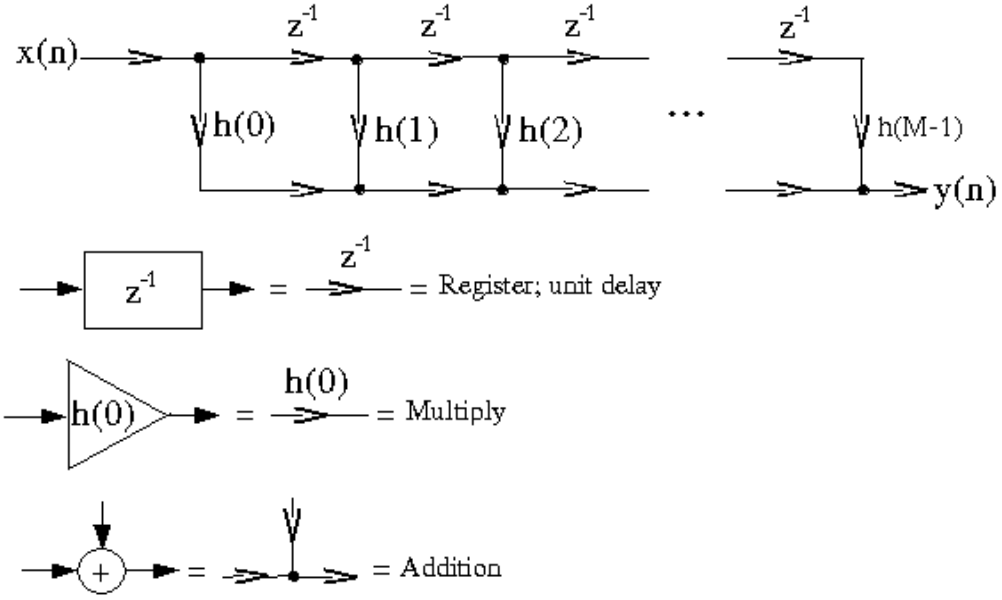
Whether FIR or IIR, a given transfer function can be implemented with many different filter structures. With infinite-precision data, coefficients, and arithmetic, all filter structures implementing the same transfer function produce the same output. However, different filter strucures may produce very different errors with quantized data and finite-precision or fixed-point arithmetic. The computational expense and memory usage may also differ greatly. Knowledge of different filter structures allows DSP engineers to trade off these factors to create the best implementation.

FIR Filter Structures

Consider causal FIR filters: $y(n) = \sum_{k=0}^{M-1} h(k)x(n-k)$; this can be realized using the following structure



or in a different notation



This is called the **direct-form FIR filter structure**.

There are no closed loops (no feedback) in this structure, so it is called a **non-recursive structure**. Since any FIR filter can be implemented using the direct-form, non-recursive structure, it is always possible to implement an FIR filter non-recursively. However, it is also possible to implement an FIR filter **recursively**, and for some special sets of FIR filter coefficients this is much more efficient.

Example:

$$y(n) = \sum_{k=0}^{M-1} x(n-k)$$

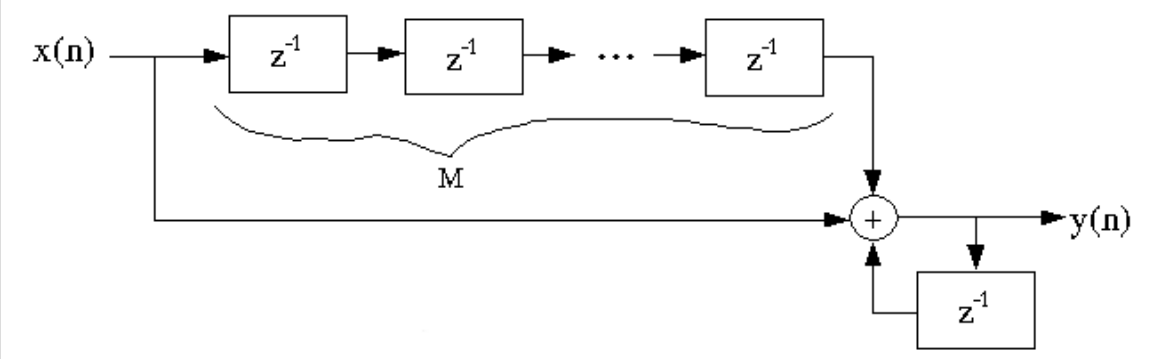
where

$$h(k) = \left\{ 0, 0, \underset{\substack{\uparrow \\ k=0}}{1}, 1, \ldots, 1, \underset{\substack{\uparrow \\ k=M-1}}{1}, 0, 0, 0, \ldots \right\}$$

But note that

$$y(n) = y(n-1) + x(n) - x(n-M)$$

This can be implemented as



Instead of costing $M-1$ adds/output point, this comb filter costs only two adds/output.

Exercise:

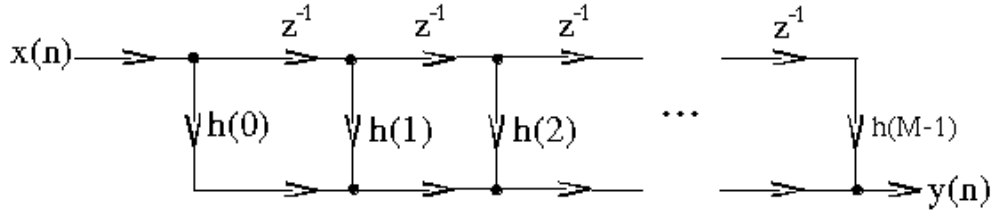
Problem: Is this stable, and if not, how can it be made so?

IIR filters must be implemented with a **recursive** structure, since that's the only way a finite number of elements can generate an infinite-length impulse response in a linear, time-invariant (LTI) system. Recursive structures have the advantages of being able to implement IIR systems, and sometimes greater computational efficiency, but the disadvantages of possible instability, limit cycles, and other deletorious effects that we will study shortly.

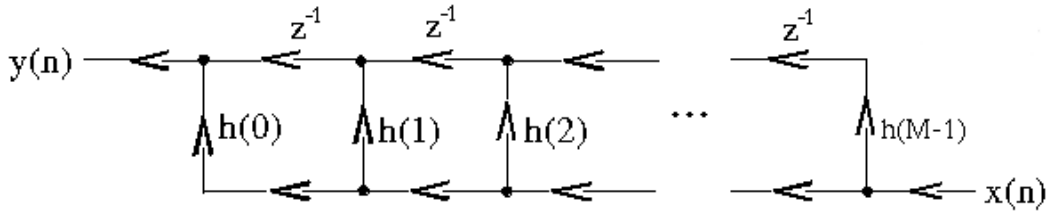
Transpose-form FIR filter structures

The **flow-graph-reversal theorem** says that if one changes the directions of all the arrows, and inputs at the output and takes the output from the input of a reversed flow-graph, the new system has an identical input-output relationship to the original flow-graph.

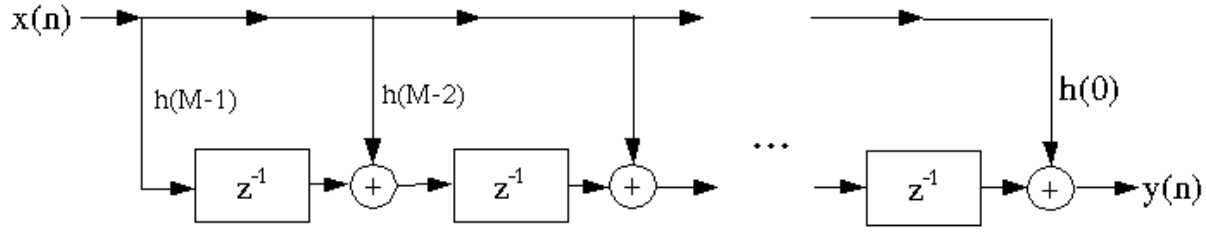
Direct-form FIR structure



reverse = transpose-form FIR filter structure



or redrawn



Cascade structures

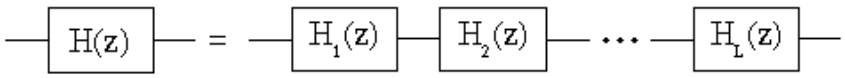
The z-transform of an FIR filter can be factored into a cascade of short-length filters

$$b_0 + b_1 z^{-1} + b_2 z^{-3} + \ldots + b_m z^{-m} = b_0 \left(1 - z_1 z^{-1}\right) \left(1 - z_2 z^{-1}\right) \ldots \left(1 - z_m z^{-1}\right)$$

where the z_i are the zeros of this polynomial. Since the coefficients of the polynomial are usually real, the roots are usually complex-conjugate pairs, so we generally combine $\left(1 - z_i z^{-1}\right) \left(1 - \overline{z_i} z^{-1}\right)$ into one quadratic (length-2) section with **real** coefficients

$$\left(1 - z_i z^{-1}\right) \left(1 - \overline{z_i} z^{-1}\right) = 1 - 2\Re(z_i) z^{-1} + \left(|z_i|\right)^2 z^{-2} = H_i(z)$$

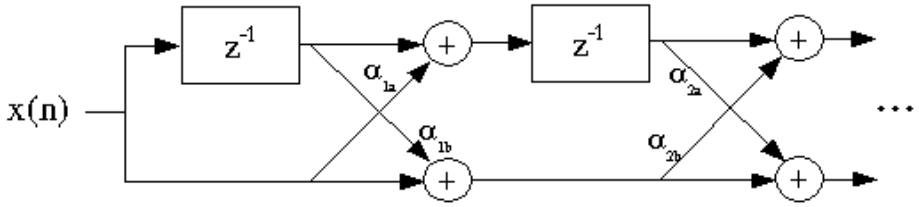
The overall filter can then be implemented in a **cascade** structure.



This is occasionally done in FIR filter implementation when one or more of the short-length filters can be implemented efficiently.

Lattice Structure

It is also possible to implement FIR filters in a lattice structure: this is sometimes used in adaptive filtering



IIR Filter Structures

IIR (Infinite Impulse Response) filter structures must be recursive (use feedback); an infinite number of coefficients could not otherwise be realized with a finite number of computations per sample.

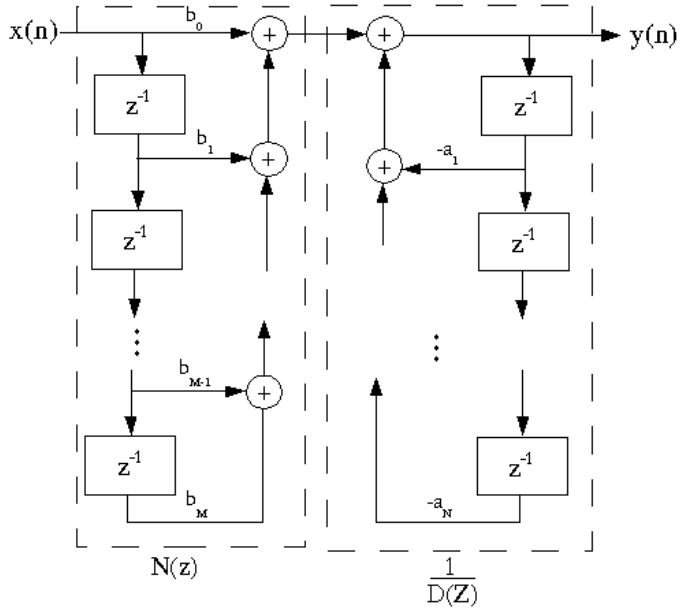
$$H(z) = \frac{N(z)}{D(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \ldots + b_M z^{-M}}{1 + a_1 z^{-1} + a_2 z^{-2} + \ldots + a_N z^{-N}}$$

The corresponding time-domain difference equation is

$$y(n) = (-(a_1 y(n-1))) - a_2 y(n-2) + \ldots - a_N y(n-N) + b_0 x(0) + b_1 x(n-1) + \ldots + b_M x(n-M)$$

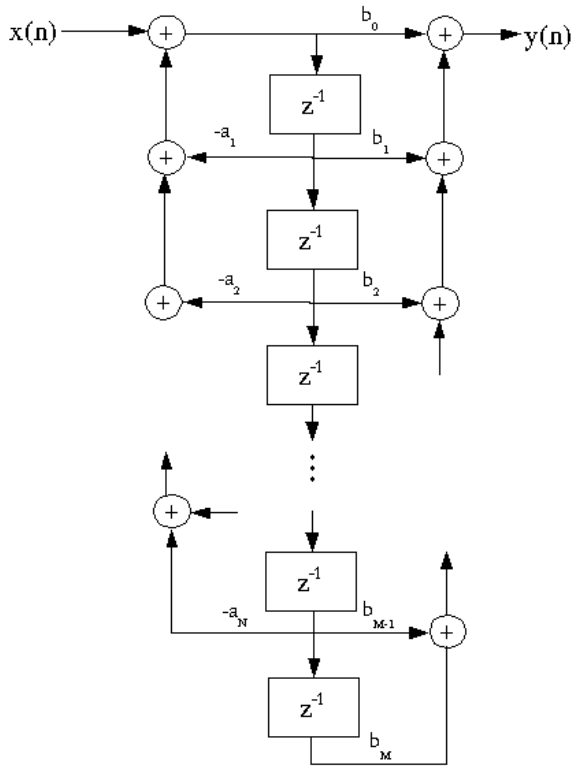
Direct-form I IIR Filter Structure

The difference equation above is implemented directly as written by the Direct-Form I IIR Filter Structure.



Note that this is a cascade of two systems, $N(z)$ and $\frac{1}{D(z)}$. If we reverse the order of the filters, the overall system is unchanged: The memory elements appear in the middle and store identical values, so they can be combined, to form the Direct-Form II IIR Filter Structure.

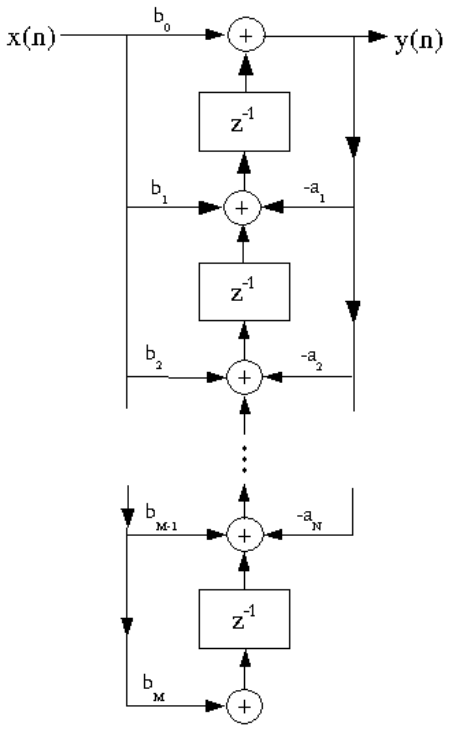
Direct-Form II IIR Filter Structure



This structure is **canonic**: (i.e., it requires the minimum number of memory elements).

Flowgraph reversal gives the

Transpose-Form IIR Filter Structure



Usually we design IIR filters with $N = M$, but not always.

Obviously, since all these structures have identical frequency response, filter structures are not unique. We consider many different structures because

1. Depending on the technology or application, one might be more convenient than another
2. The response in a practical realization, in which the data and coefficients must be **quantized**, may differ substantially, and some structures behave much better than others with quantization.

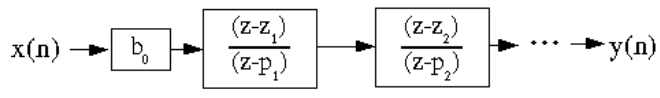
The Cascade-Form IIR filter structure is one of the least sensitive to quantization, which is why it is the most commonly used IIR filter structure.

IIR Cascade Form

The numerator and denominator polynomials can be factored

$$H(z) = \frac{b_0 + b_1 z^{-1} + \ldots + b_M z^{-m}}{1 + a_1 z^{-1} + \ldots + a_N z^{-N}} = \frac{b_0 \prod_{k=1}^{M} z - z_k}{z^{M-N} \prod_{i=1}^{N} z - p_k}$$

and implemented as a cascade of short IIR filters.



Since the filter coefficients are usually real yet the roots are mostly complex, we actually implement these as second-order sections, where comple-conjugate pole and zero pairs are combined into second-order sections

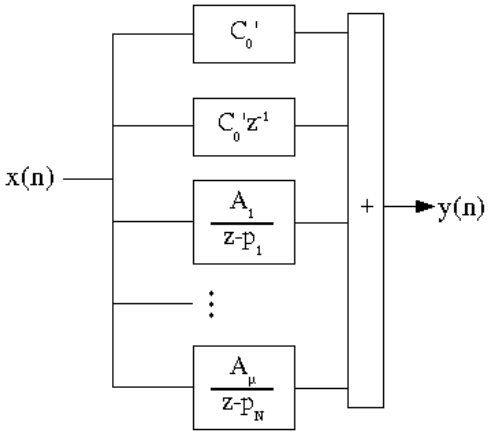
with real coefficients. The second-order sections are usually implemented with either the Direct-Form II or Transpose-Form structure.

Parallel form

A rational transfer function can also be written as

$$\frac{b_0 + b_1 z^{-1} + \ldots + b_M z^{-m}}{1 + a_1 z^{-1} + \ldots + a_N z^{-N}} = {c_0}' + {c_1}' z^{-1} + \ldots + \frac{A_1}{z - p_1} + \frac{A_2}{z - p_2} + \ldots + \frac{A_N}{z - p_N}$$

which by linearity can be implemented as



As before, we combine complex-conjugate pole pairs into second-order sections with real coefficients.

The cascade and parallel forms are of interest because they are much less sensitive to coefficient quantization than higher-order structures, as analyzed in later modules in this course.

Other forms

There are many other structures for IIR filters, such as wave digital filter structures, lattice-ladder, all-pass-based forms, and so forth. These are the result of extensive research to find structures which are computationally efficient **and** insensitive to quantization error. They all represent various tradeoffs; the best choice in a given context is not yet fully understood, and may never be.

State-Variable Representation of Discrete-Time Systems

State and the State-Variable Representation

State
: the minimum additional information at time n, which, along with all current and future input values, is necessary to compute all future outputs.

Essentially, the state of a system is the information held in the delay registers in a filter structure or signal flow graph.

Note: Any LTI (linear, time-invariant) system of finite order M can be represented by a state-variable description

$$\boldsymbol{x}(n+1) = A\boldsymbol{x}(n) + \boldsymbol{B}u(n)$$

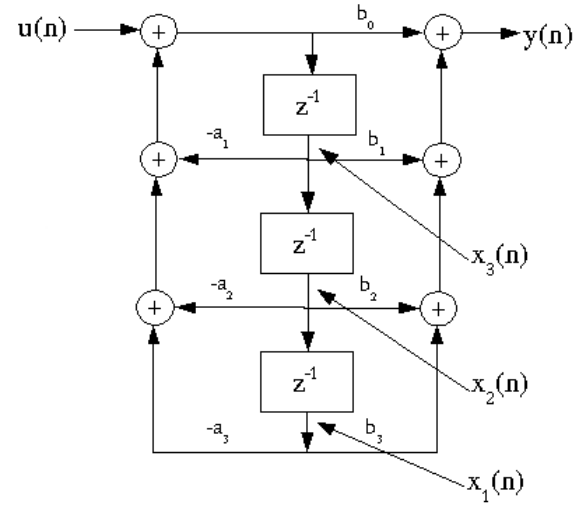
$$\boldsymbol{y}(n) = \boldsymbol{C}\boldsymbol{x}(n) + Du(n)$$

where $\boldsymbol{x}$ is an M x 1 "state vector," $u(n)$ is the input at time n, $y(n)$ is the output at time n; A is an M x M matrix, $\boldsymbol{B}$ is an M x 1 vector, $\boldsymbol{C}$ is a 1 x M vector, and D is a 1 x 1 scalar.

One can always obtain a state-variable description of a signal flow graph.

Example:
3rd-Order IIR

$y(n) = (-(a_1 y(n-1))) - a_2 y(n-2) - a_3 y(n-3) + b_0 x(n) + b_1 x(n-1) + b_2 x(n-2) + b_3 x(n-3)$



$$\begin{pmatrix} x_1(n+1) \\ x_2(n+1) \\ x_3(n+1) \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -a_3 & -a_2 & -a_1 \end{pmatrix} \begin{pmatrix} x_1(n) \\ x_2(n) \\ x_3(n) \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} u(n)$$

$$y(n) = \begin{pmatrix} -(a_3 b_0) & -(a_2 b_0) & -(a_1 b_0) \end{pmatrix} \begin{pmatrix} x_1(n) \\ x_2(n) \\ x_3(n) \end{pmatrix} + (b_0) u(n)$$

Exercise:

Problem: Is the state-variable description of a filter $H(z)$ unique?

Exercise:

Problem: Does the state-variable description fully describe the signal flow graph?

State-Variable Transformation

Suppose we wish to define a new set of state variables, related to the old set by a linear transformation: $\boldsymbol{q}(n) = T\boldsymbol{x}(n)$, where T is a nonsingular M x M matrix, and $\boldsymbol{q}(n)$ is the new state vector. We wish the overall system to remain the same. Note that $\boldsymbol{x}(n) = T^{-1}\boldsymbol{q}(n)$, and thus

$$\boldsymbol{x}(n+1) = A\boldsymbol{x}(n) + Bu(n) \Rightarrow T^{-1}\boldsymbol{q}(n) = AT^{-1}\boldsymbol{q}(n) + Bu(n) \Rightarrow \boldsymbol{q}(n) = TAT^{-1}\boldsymbol{q}(n) + TBu(n)$$

$$y(n) = C\boldsymbol{x}(n) + Du(n) \Rightarrow y(n) = CT^{-1}\boldsymbol{q}(n) + Du(n)$$

This defines a new state system with an input-output behavior identical to the old system, but with different internal memory contents (states) and state matrices.

$$\boldsymbol{q}(n) = \hat{A}\boldsymbol{q}(n) + \hat{B}u(n)$$

$$y(n) = \hat{C}\boldsymbol{q}(n) + \hat{D}u(n)$$

$\hat{A} = TAT^{-1}$, $\hat{B} = TB$, $\hat{C} = CT^{-1}$, $\hat{D} = D$

These transformations can be used to generate a wide variety of alternative stuctures or implementations of a filter.

Transfer Function and the State-Variable Description

Taking the z transform of the state equations

$$Z[\boldsymbol{x}(n+1)] = Z[A\boldsymbol{x}(n) + Bu(n)]$$

$$Z[y(n)] = Z[C\boldsymbol{x}(n) + Du(n)]$$

$$\Downarrow$$

$$z\boldsymbol{X}(z) = A\boldsymbol{X}(z) + BU(z)$$

Note: $\boldsymbol{X}(z)$ is a vector of scalar z-transforms $\boldsymbol{X}(z)^T = \begin{pmatrix} X_1(z) & X_2(z) & \dots \end{pmatrix}$

$$\boldsymbol{Y}(z) = C\boldsymbol{X}(n) + DU(n)$$

$$(zI - A)\boldsymbol{X}(z) = BU(z) \Rightarrow \boldsymbol{X}(z) = (zI - A)^{-1}BU(z)$$

so
Equation:

$$\begin{aligned} Y(z) &= C(zI - A)^{-1}BU(z) + DU(z) \\ &= \Big(C(-(zI))^{-1}B + D\Big)U(z) \end{aligned}$$

and thus

$$H(z) = C(zI - A)^{-1}B + D$$

Note that since $(zI - A)^{-1} = \frac{\pm(\det{(zI-A)_{\text{red}}})^T}{\det(z(I)-A)}$, this transfer function is an Mth-order rational fraction in z. The denominator polynomial is $D(z) = \det{(zI - A)}$. A discrete-time state system is thus stable if the M roots of $\det{(zI - A)}$ (i.e., the poles of the digital filter) are all inside the unit circle.

Consider the transformed state system with $\hat{A} = TAT^{-1}$, $\hat{B} = TB$, $\hat{C} = CT^{-1}$, $\hat{D} = D$:
Equation:

$$\begin{aligned} H(z) &= \hat{C}\Big(zI - \hat{A}\Big)^{-1}\hat{B} + \hat{D} \\ &= CT^{-1}\big(zI - TAT^{-1}\big)^{-1}TB + D \\ &= CT^{-1}\big(T\,(zI - A)T^{-1}\big)^{-1}TB + D \\ &= CT^{-1}\big(T^{-1}\big)^{-1}(zI - A)^{-1}T^{-1}TB + D \\ &= C(zI - A)^{-1}B + D \end{aligned}$$

This proves that state-variable transformation doesn't change the transfer function of the underlying system. However, it can provide alternate forms that are less sensitive to coefficient quantization or easier to analyze, understand, or implement.

State-variable descriptions of systems are useful because they provide a fairly general tool for analyzing all systems; they provide a more detailed description of a signal flow graph than does the transfer function (although not a full description); and they suggest a large class of alternative implementations. They are even more useful in control theory, which is largely based on state descriptions of systems.

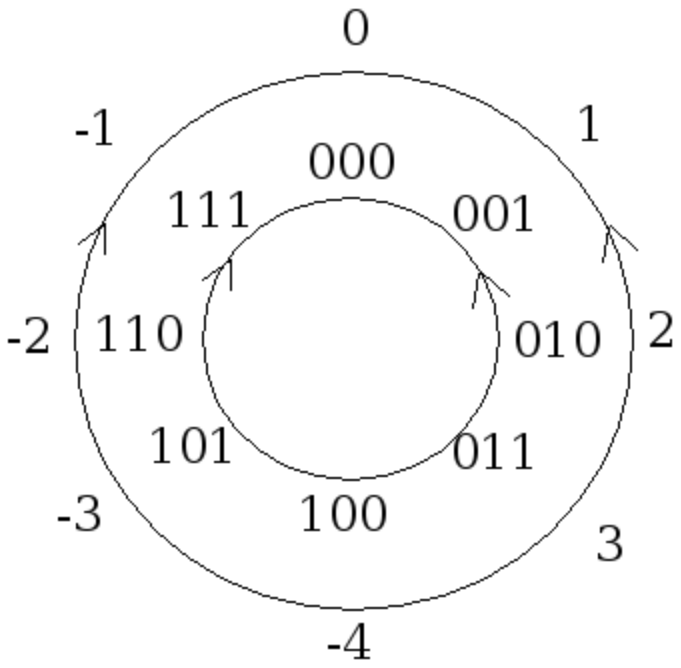
Fixed-Point Number Representation

Fixed-point arithmetic is generally used when hardware cost, speed, or complexity is important. Finite-precision quantization issues usually arise in fixed-point systems, so we concentrate on fixed-point quantization and error analysis in the remainder of this course. For basic signal processing computations such as digital filters and FFTs, the magnitude of the data, the internal states, and the output can usually be scaled to obtain good performance with a fixed-point implementation.

Two's-Complement Integer Representation

As far as the hardware is concerned, fixed-point number systems represent data as B-bit integers. The two's-complement number system is usually used:

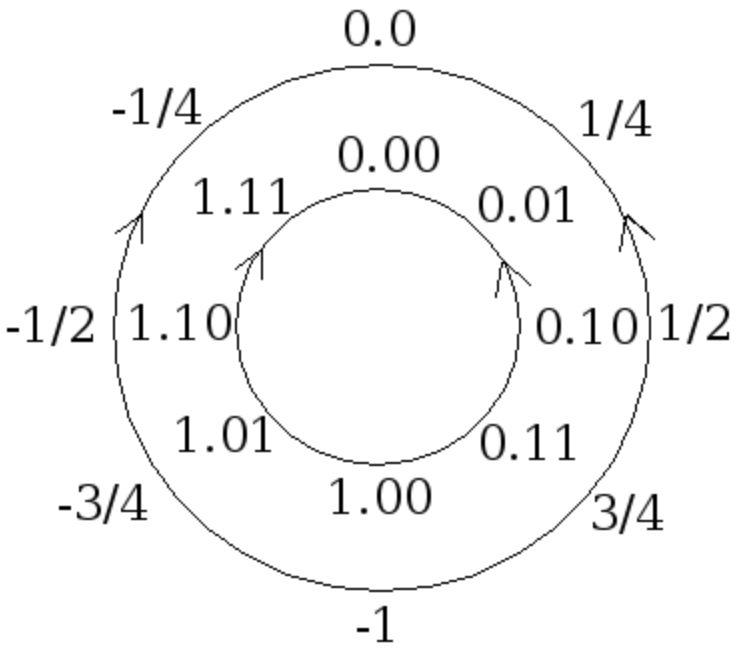
$$k = \begin{cases} \text{binary integer representation} \;\; \text{if} \;\; 0 \leq \text{k} \leq 2^{B-1} - 1 \\ \text{bit-by-bit inverse}(-k) + 1 \;\; \text{if} \;\; -2^{B-1} \leq k \leq 0 \end{cases}$$



The most significant bit is known at the **sign bit**; it is 0 when the number is non-negative; 1 when the number is negative.

Fractional Fixed-Point Number Representation

For the purposes of signal processing, we often regard the fixed-point numbers as binary fractions between $[-1, 1)$, by implicitly placing a decimal point after the sign bit.



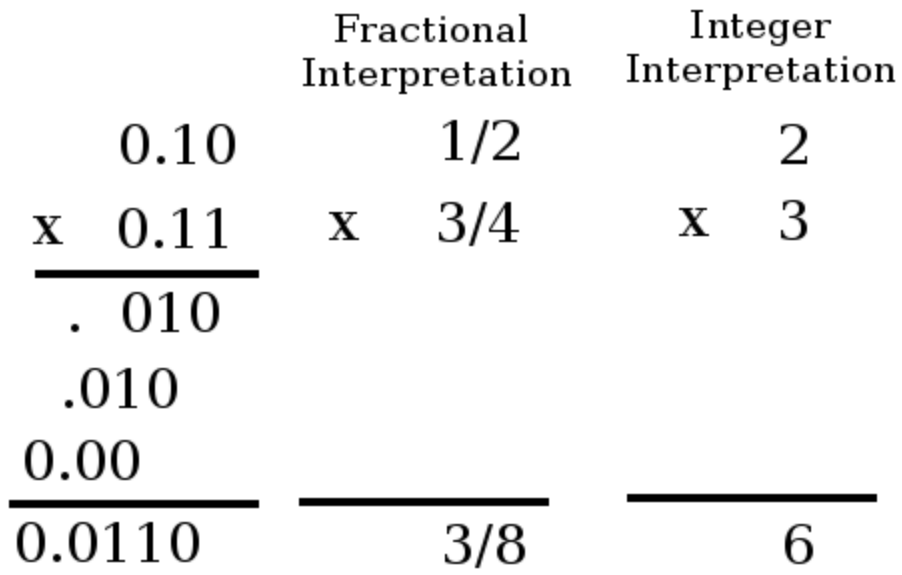
or

$$x = -b_0 + \sum_{i=1}^{B-1} b_i 2^{-i}$$

This interpretation makes it clearer how to implement digital filters in fixed-point, at least when the coefficients have a magnitude less than 1.

Truncation Error

Consider the multiplication of two binary fractions



Note that full-precision multiplication almost doubles the number of bits; if we wish to return the product to a B-bit representation, we must truncate the $B-1$ least significant bits. However, this introduces **truncation error** (also known as **quantization error**, or **roundoff error** if the number is rounded to the nearest B-bit fractional value rather than truncated). Note that this occurs after **multiplication**.

Overflow Error

Consider the addition of two binary fractions;

	Fractional Interpretation	Integer Interpretation
0.10	1/2	2
+ 0.11	+ 3/4	+ 3
1.01	5/4 = -1/4	5 = -1

Note the occurence of wraparound **overflow**; this only happens with **addition**. Obviously, it can be a bad problem.

There are thus two types of fixed-point error: roundoff error, associated with data quantization and multiplication, and overflow error, associated with data quantization and additions. In fixed-point systems, one must strike a balance between these two error sources; by scaling down the data, the occurence of overflow errors is reduced, but the relative size of the roundoff error is increased.

Note: Since multiplies require a number of additions, they are especially expensive in terms of hardware (with a complexity proportional to $B_x B_h$, where B_x is the number of bits in the data, and B_h is the number of bits in the filter coefficients). Designers try to minimize both B_x and B_h, and often choose $B_x \neq B_h$!

Fixed-Point Quantization

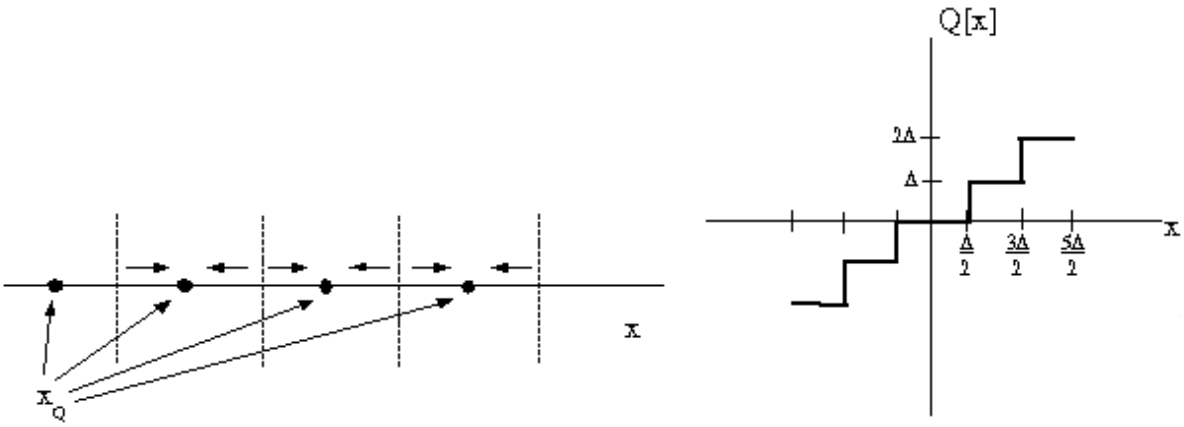
The fractional B-bit two's complement number representation evenly distributes 2^B quantization levels between -1 and $1 - 2^{-(B-1)}$. The spacing between quantization levels is then

$$\frac{2}{2^B} = 2^{-(B-1)} \doteq \Delta_B$$

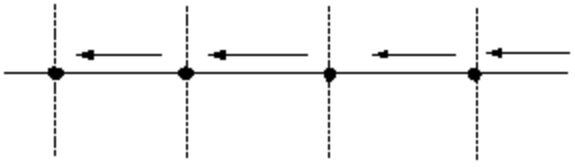
Any signal value falling between two levels is assigned to one of the two levels.

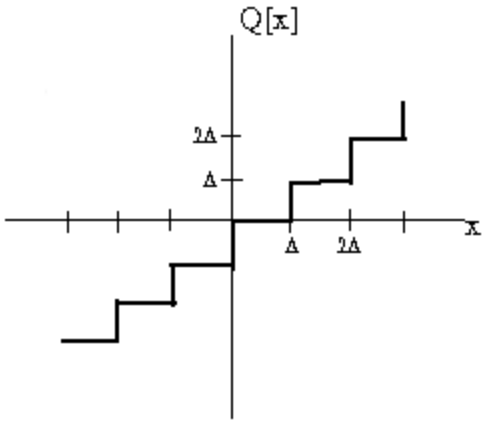
$X_Q = Q[x]$ is our notation for quantization. $e = Q[x] - x$ is then the quantization error.

One method of quantization is **rounding**, which assigns the signal value to the **nearest** level. The maximum error is thus $\frac{\Delta_B}{2} = 2^{-B}$.



Another common scheme, which is often easier to implement in hardware, is **truncation**. $Q[x]$ assigns x to the next lowest level.

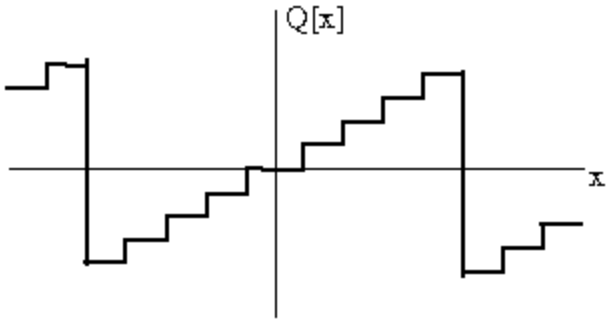




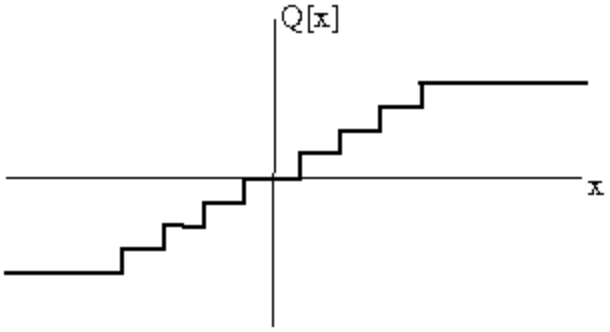
The worst-case error with truncation is $\Delta = 2^{-(B-1)}$, which is twice as large as with rounding. Also, the error is always negative, so on average it may have a non-zero mean (i.e., a bias component).

Overflow is the other problem. There are two common types: two's complement (or **wraparound**) overflow, or **saturation** overflow.

wraparound



saturation



Obviously, overflow errors are bad because they are typically **large**; two's complement (or wraparound) overflow introduces more error than

saturation, but is easier to implement in hardware. It also has the advantage that if the **sum** of several numbers is between $[-1, 1)$, the final answer will be correct even if intermediate sums overflow! However, wraparound overflow leaves IIR systems susceptible to zero-input large-scale limit cycles, as discussed in another module. As usual, there are many tradeoffs to evaluate, and no one right answer for all applications.

Finite-Precision Error Analysis

Fundamental Assumptions in finite-precision error analysis

Quantization is a highly nonlinear process and is very difficult to analyze precisely. Approximations and assumptions are made to make analysis tractable.

Assumption #1

The roundoff or truncation errors at any point in a system at each time are **random**, **stationary**, and **statistically independent** (white and independent of all other quantizers in a system).

That is, the error autocorrelation function is $r_e[k] = E[e_n e_{n+k}] = {\sigma_q}^2 \delta[k]$. Intuitively, and confirmed experimentally in some (but not all!) cases, one expects the quantization error to have a uniform distribution over the interval $\left[-\frac{\Delta}{2}, \frac{\Delta}{2}\right)$ for rounding, or $(-\Delta, 0]$ for truncation.

In this case, rounding has zero mean and variance

$$E[Q[x_n] - x_n] = 0$$

$${\sigma_Q}^2 = E\left[{e_n}^2\right] = \frac{{\Delta_B}^2}{12}$$

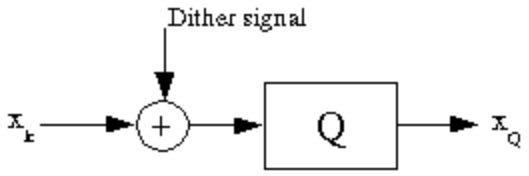
and truncation has the statistics

$$E[Q[x_n] - x_n] = -\frac{\Delta}{2}$$

$${\sigma_Q}^2 = \frac{{\Delta_B}^2}{12}$$

Please note that the independence assumption may be very bad (for example, when quantizing a sinusoid with an integer period N). There is

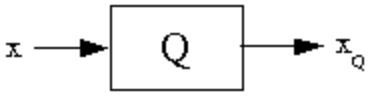
another quantizing scheme called **dithering**, in which the values are randomly assigned to nearby quantization levels. This can be (and often is) implemented by adding a small (one- or two-bit) random input to the signal before a truncation or rounding quantizer.



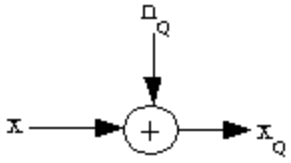
This is used extensively in practice. Altough the overall error is somewhat higher, it is spread evenly over all frequencies, rather than being concentrated in spectral lines. This is very important when quantizing sinusoidal or other periodic signals, for example.

Assumption #2

Pretend that the quantization error is really additive **Gaussian** noise with the same mean and variance as the uniform quantizer. That is, model



as



This model is a **linear** system, which our standard theory can handle easily. We model the noise as Gaussian because it remains Gaussian after passing through filters, so analysis in a system context is tractable.

Summary of Useful Statistical Facts

- **correlation function** $r_x[k] \doteq E[x_n x_{n+k}]$
- **power spectral density** $S_x(w) \doteq \mathrm{DTFT}\,[r_x[n]]$
- Note $r_x[0] = {\sigma_x}^2 = \frac{1}{2\pi}\int_{-\pi}^{\pi} S_x(w)\ \mathrm{d}\ w$
- $r_{\mathrm{xy}}[k] \doteq E\left[x^*[n]y[n+k]\right]$
- **cross-spectral density** $S_{\mathrm{xy}}(w) = \mathrm{DTFT}\,[r_{\mathrm{xy}}[n]]$
- For $y = h^* x$:

$$S_{\mathrm{yx}}(w) = H(w)S_x(w)$$

$$S_{\mathrm{yy}}(w) = (|H(w)|)^2 S_x(w)$$

- Note that the **output** noise level after filtering a noise sequence is

$$ {\sigma_y}^2 = r_{\mathrm{yy}}[0] = \frac{1}{\pi}\int_{-\pi}^{\pi} (|H(w)|)^2 S_x(w)\ \mathrm{d}\ w$$

  so postfiltering quantization noise alters the noise power spectrum and may change its variance!
- For x_1, x_2 statistically independent

$$r_{x_1+x_2}[k] = r_{x_1}[k] + r_{x_2}[k]$$

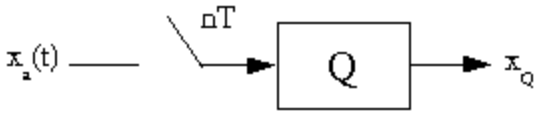
$$S_{x_1+x_2}(w) = S_{x_1}(w) + S_{x_2}(w)$$

- For independent random variables

$$ {\sigma_{x_1+x_2}}^2 = {\sigma_{x_1}}^2 + {\sigma_{x_2}}^2$$

Input Quantization Noise Analysis

All practical analog-to-digital converters (A/D) must quantize the input data. This can be modeled as an ideal sampler followed by a B-bit quantizer.



The signal-to-noise ratio (SNR) of an A/D is
Equation:

$$\begin{aligned} \mathrm{SNR} &= 10 \log \frac{P_x}{P_n} \\ &= 10 \log P_x - 10 \log \frac{{\Delta_B}^2}{12} \\ &= 10 \log P_x + 4.77 + 6.02B \end{aligned}$$

where P_x is the power in the signal and P_n is the power of the quantization noise, which equals its variance if it has a zero mean. The SNR increases by 6dB with each additional bit.

Quantization Error in FIR Filters

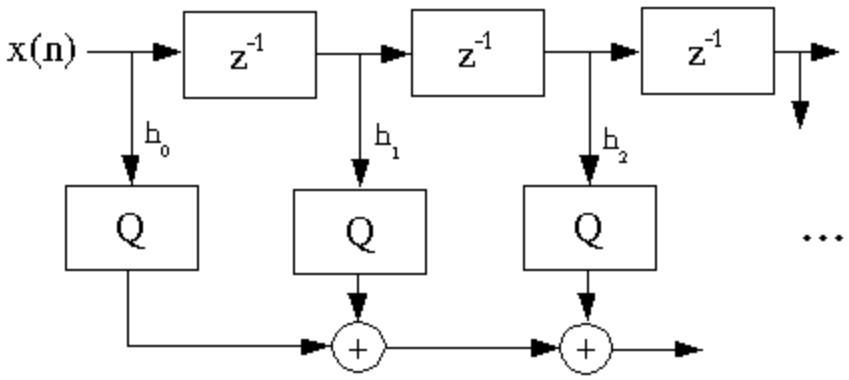
In digital filters, both the data at various places in the filter, which are continually varying, and the coefficients, which are fixed, must be quantized. The effects of quantization on data and coefficients are quite different, so they are analyzed separately.

Data Quantization

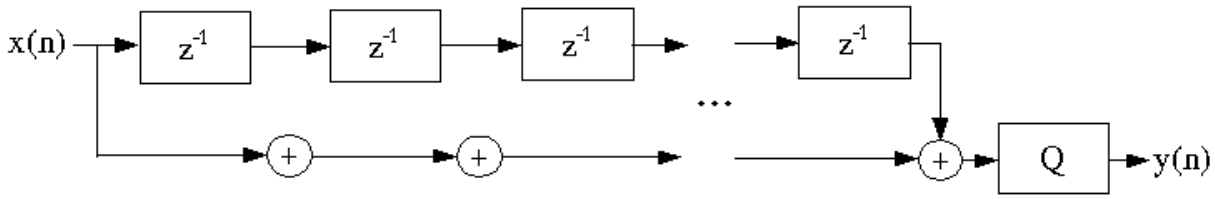
Typically, the input and output in a digital filter are quantized by the analog-to-digital and digital-to-analog converters, respectively. Quantization also occurs at various points in a filter structure, usually after a multiply, since multiplies increase the number of bits.

Direct-form Structures

There are two common possibilities for quantization in a direct-form FIR filter structure: after each multiply, or only once at the end.



Single-precision accumulate; total variance —

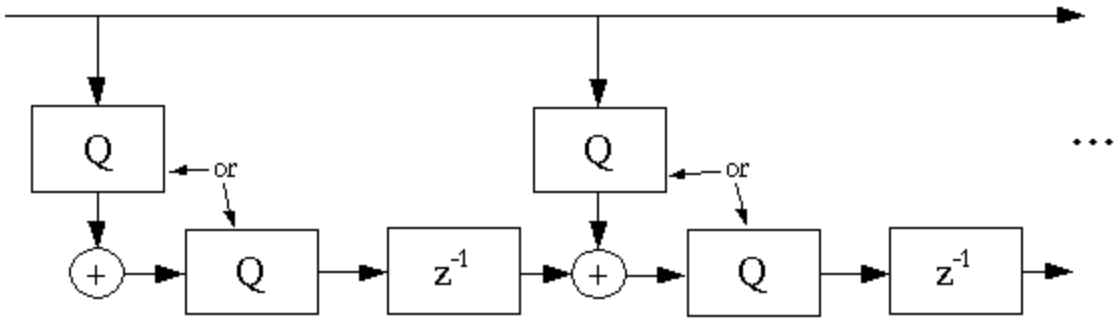


Double-precision accumulate; variance —

In the latter structure, a double-length accumulator adds all bits of each product into the accumulating sum, and truncates only at the end. Obviously, this is much preferred, and should **always** be used wherever possible. All DSP microprocessors and most general-pupose computers support double-precision accumulation.

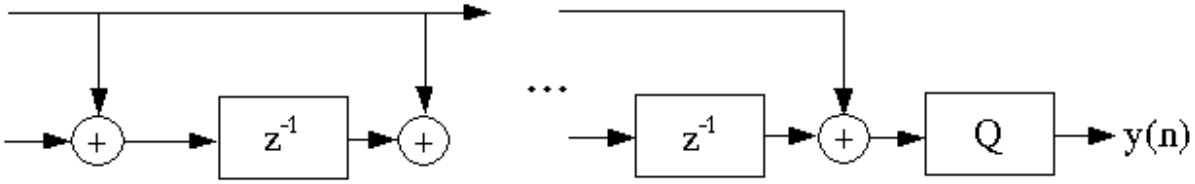
Transpose-form

Similarly, the transpose-form FIR filter structure presents two common options for quantization: after each multiply, or once at the end.



Quantize at each stage before storing intermediate sum. Output variance —

or



Store double-precision partial sums. Costs more memory, but variance

—

The transpose form is not as convenient in terms of supporting double-precision accumulation, which is a significant disadvantage of this structure.

Coefficient Quantization

Since a quantized coefficient is fixed for all time, we treat it differently than data quantization. The fundamental question is: how much does the quantization affect the frequency response of the filter?

The quantized filter frequency response is

Assuming the quantization model is correct, should be fairly random and white, with the error spread fairly equally over all frequencies ; however, the randomness of this error destroys any equiripple property or any infinite-precision optimality of a filter.

Exercise:

Problem:

What quantization scheme minimizes the quantization error in frequency (minimizes)? On average, how big is this error?

Ideally, if one knows the coefficients are to be quantized to bits, one should incorporate this directly into the filter design problem, and find the -bit binary fractional coefficients minimizing the maximum deviation (error). This can be done, but it is an integer program, which is known to be np-hard (i.e., requires almost a brute-force search). This is so expensive computationally that it's rarely done. There are some sub-optimal methods that are much more efficient and usually produce pretty good results.

Data Quantization in IIR Filters

Finite-precision effects are much more of a concern with IIR filters than with FIR filters, since the effects are more difficult to analyze and minimize, coefficient quantization errors can cause the filters to become unstable, and disastrous things like large-scale limit cycles can occur.

Roundoff noise analysis in IIR filters

Suppose there are several quantization points in an IIR filter structure. By our simplifying assumptions about quantization error and Parseval's theorem, the quantization noise variance at the output of the filter from the th quantizer is

Equation:

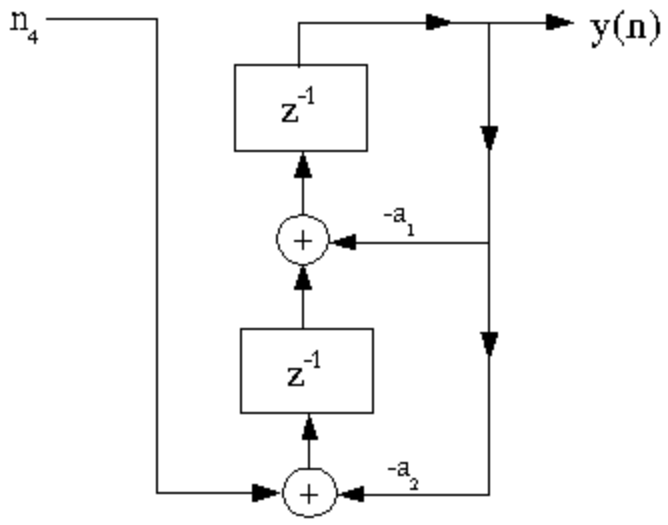
—

—

where is the variance of the quantization error at the th quantizer,
is the power spectral density of that quantization error, and
is the transfer function from the th quantizer to the output point.
Thus for independent quantizers in the structure, the total quantization noise variance is

—

Note that in general, each , and thus the variance at the output due to each quantizer, is different; for example, the system as seen by a quantizer at the input to the first delay state in the Direct-Form II IIR filter structure to the output, call it , is

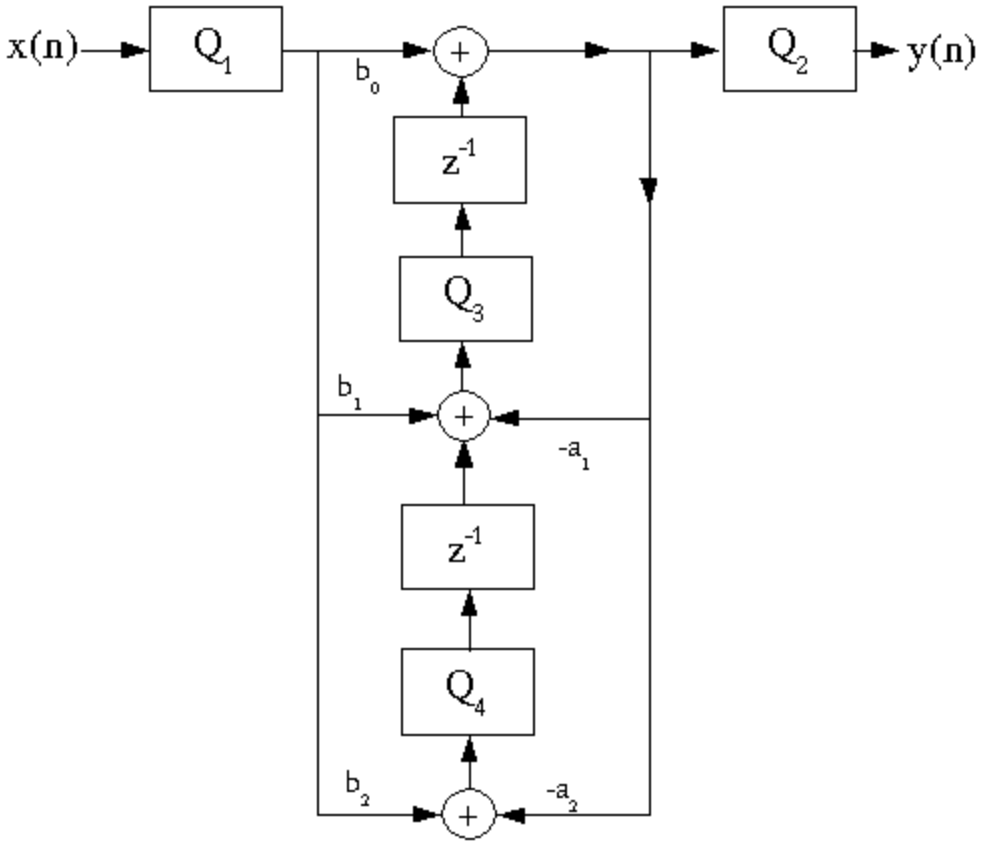


with a transfer function

---

which can be evaluated at to obtain the frequency response.

A general approach to find is to write state equations for the equivalent structure as seen by , and to determine the transfer function according to .

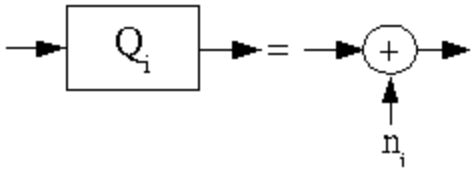


Exercise:

Problem:

The above figure illustrates the quantization points in a typical implementation of a Direct-Form II IIR second-order section. What is the total variance of the output error due to all of the quantizers in the system?

By making the assumption that each represents a noise source that is white, independent of the other sources, and additive,



the variance at the output is the sum of the variances at the output due to each noise source:

The variance due to each noise source at the output can be determined from — ; note that by our assumptions, and is the transfer function **from the noise source to the output**.

IIR Coefficient Quantization Analysis

Coefficient quantization is an important concern with IIR filters, since straigthforward quantization often yields poor results, and because quantization can produce unstable filters.

Sensitivity analysis

The performance and stability of an IIR filter depends on the pole locations, so it is important to know how quantization of the filter coefficients a_k affects the pole locations p_j. The denominator polynomial is

$$D\ z \qquad \begin{matrix} N \\ k \end{matrix} \ a_k z^{\ k} \qquad \begin{matrix} N \\ i \end{matrix} \ p_i z$$

We wish to know $\frac{p_i}{a_k}$, which, for small deviations, will tell us that a δ change in a_k yields an $\varepsilon \quad \delta \frac{p_i}{a_k}$ change in the pole location. $\frac{p_i}{a_k}$ is the **sensitivity** of the pole location to quantization of a_k. We can find $\frac{p_i}{a_k}$ using the chain rule.

$$\frac{A\ z}{a_k}\ _{z\ \ p_i} \qquad \frac{A\ z}{z}\frac{z}{a_k}\ _{z\ \ p_i}$$

$$\frac{p_i}{a_k} \qquad \frac{\frac{A\ z_i}{a_k}\ _{z\ \ p_i}}{\frac{A\ z_i}{z}\ _{z\ \ p_i}}$$

which is
Equation:

$$\frac{p_i}{a_k} \qquad \frac{z^{\ k}}{z \quad {}^{N}_{j\ j\ i} \quad p_j z} \quad z \quad p_i$$

$$\frac{p_i{}^{N\ k}}{{}^{N}_{j\ j\ i}\ p_j \quad p_i}$$

Note that as the poles get closer together, the sensitivity increases greatly. So as the filter order increases and more poles get stuffed closer together inside the unit circle, the error introduced by coefficient quantization in the pole locations grows rapidly.

How can we reduce this high sensitivity to IIR filter coefficient quantization?

Solution

Cascade or parallel form implementations! The numerator and denominator polynomials can be factored off-line at very high precision and grouped into second-order sections, which are then quantized section by section. The sensitivity of the quantization is thus that of second-order, rather than N-th order, polynomials. This yields major improvements in the frequency response of the overall filter, and is almost always done in practice.

Note that the numerator polynomial faces the same sensitivity issues; the **cascade** form also improves the sensitivity of the zeros, because they are also factored into second-order terms. However, in the **parallel** form, the zeros are globally distributed across the sections, so they suffer from quantization of all the blocks. Thus the **cascade** form preserves zero locations much better than the parallel form, which typically means that the stopband behavior is better in the cascade form, so it is most often used in practice.

Note: On the basis of the preceding analysis, it would seem important to use cascade structures in FIR filter implementations. However, most FIR

filters are linear-phase and thus symmetric or anti-symmetric. As long as the quantization is implemented such that the filter coefficients retain symmetry, the filter retains linear phase. Furthermore, since all zeros off the unit circle must appear in groups of four for symmetric linear-phase filters, zero pairs can leave the unit circle only by joining with another pair. This requires relatively severe quantizations (enough to completely remove or change the sign of a ripple in the amplitude response). This "reluctance" of pole pairs to leave the unit circle tends to keep quantization from damaging the frequency response as much as might be expected, enough so that cascade structures are rarely used for FIR filters.

Exercise:

Problem: What is the worst-case pole pair in an IIR digital filter?

Solution:

The pole pair closest to the real axis in the z-plane, since the complex-conjugate poles will be closest together and thus have the highest sensitivity to quantization.

Quantized Pole Locations

In a direct-form or transpose-form implementation of a second-order section, the filter coefficients are quantized versions of the polynomial coefficients.

$$D\ z \quad z \quad a\ z \quad a \quad z \quad p \quad z \quad p$$

$$p \quad \frac{a \quad \overline{a \quad a}}{}$$

$$p \quad re^{i\theta}$$

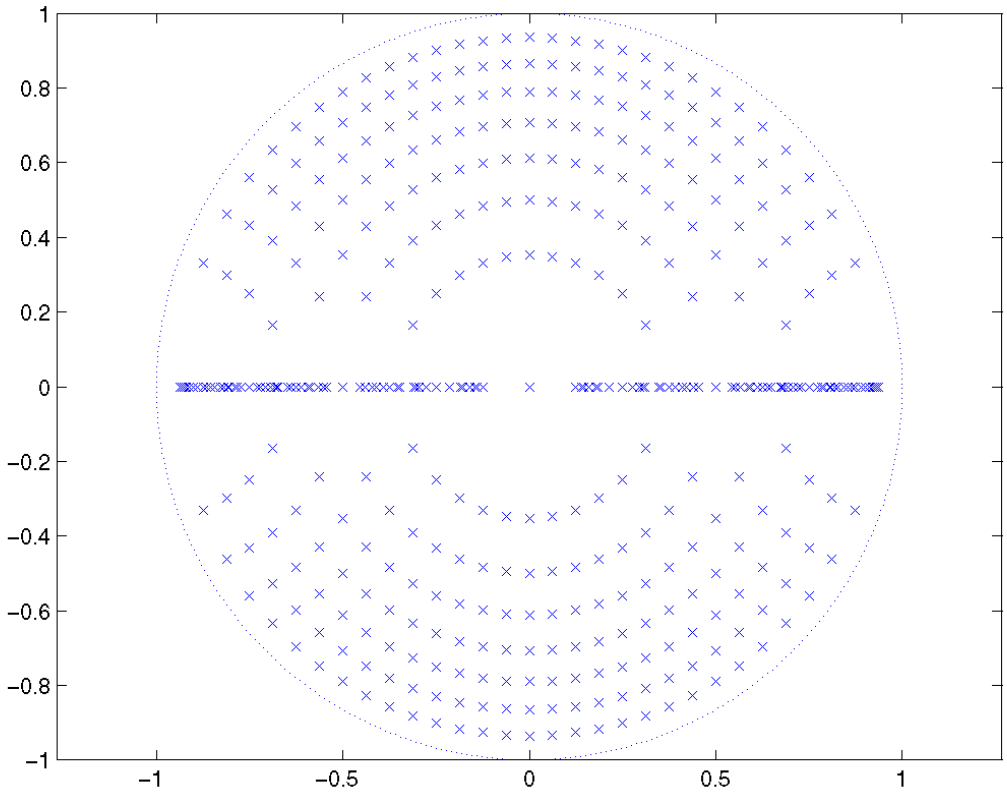
$$D\ z \quad z \quad r \quad \theta \quad r$$

So

$$a \qquad r \qquad \theta$$

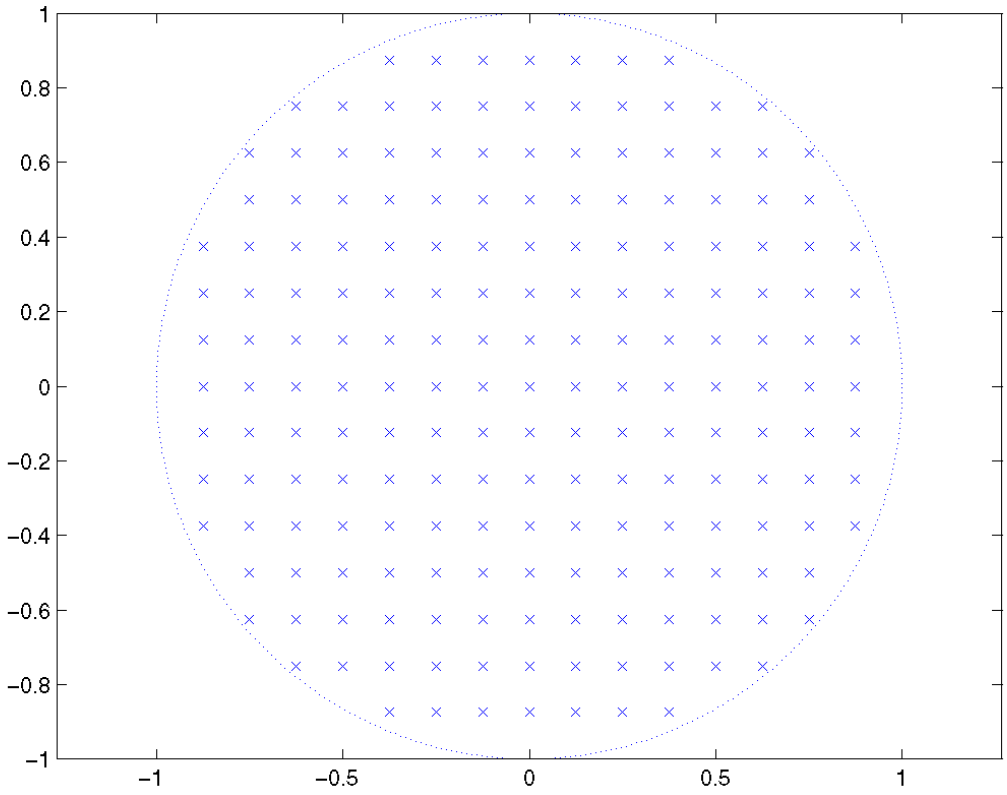
$$a \qquad r$$

Thus the quantization of a and a to B bits restricts the radius r to r $\overline{k\Delta_B}$, and a p $k\Delta_B$ The following figure shows all stable pole locations after four-bit two's-complement quantization.



Note the nonuniform distribution of possible pole locations. This might be **good** for poles near r , θ $\frac{\pi}{}$, but not so good for poles near the origin or the Nyquist frequency.

In the "normal-form" structures, a state-variable based realization, the poles are uniformly spaced.



This can only be accomplished if the coefficients to be quantized equal the real and imaginary parts of the pole location; that is,

$$\alpha \quad r \quad \theta \quad r$$

$$\alpha \quad r \quad \theta \quad p$$

This is the case for a 2nd-order system with the state matrix

A $\begin{matrix} \alpha & \alpha \\ \alpha & \alpha \end{matrix}$: The denominator polynomial is

Equation:

$$zI \quad A \qquad z \quad \alpha \qquad \alpha$$
$$z \qquad \alpha \; z \quad \alpha \qquad \alpha$$
$$z \qquad r \qquad \theta \; z \quad r \qquad \theta \qquad \theta$$
$$z \qquad r \qquad \theta \; z \quad r$$

Given any second-order filter coefficient set, we can write it as a state-space system, find a transformation matrix T such that $A \quad T \quad AT$ is in normal form, and then implement the second-order section using a structure corresponding to the state equations.

The normal form has a number of other advantages; both eigenvalues are equal, so it minimizes the norm of Ax, which makes overflow less likely, and it minimizes the output variance due to quantization of the state values. It is sometimes used when minimization of finite-precision effects is critical.

Exercise:

Problem: What is the disadvantage of the normal form?

Solution:

It requires more computation. The general state-variable equation requires nine multiplies, rather than the five used by the Direct-Form II or Transpose-Form structures.

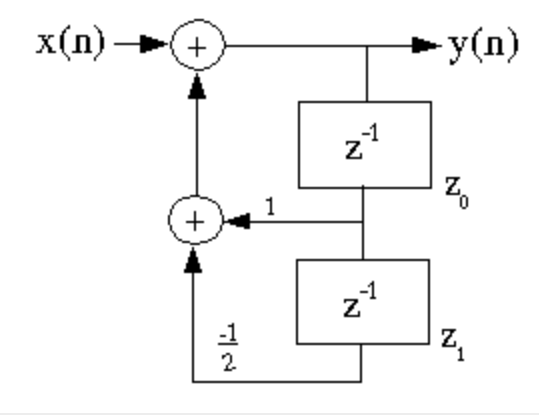
Limit Cycles

Large-scale limit cycles

When overflow occurs, even otherwise stable filters may get stuck in a **large-scale limit cycle**, which is a short-period, almost full-scale persistent filter output caused by overflow.

Example:
Consider the second-order system

$$H(z) = \frac{\quad}{z \quad -z}$$

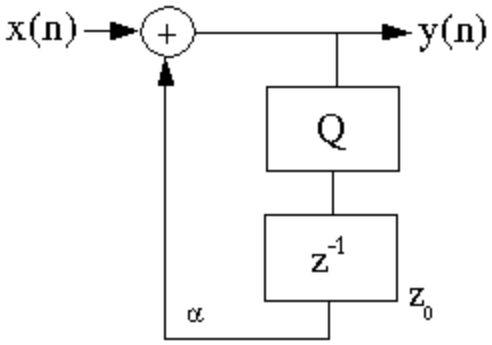


with zero input and initial state values z , z . Note $y\,n$ $z\ n$.
The filter is obviously stable, since the magnitude of the poles is — , which is well inside the unit circle. However, with wraparound overflow, note that y z — — — — —, and that z y — ——— —, so $y\,n$ — — — — even with zero input.

Clearly, such behavior is intolerable and must be prevented. Saturation arithmetic has been proved to prevent **zero-input limit cycles**, which is one reason why all DSP microprocessors support this feature. In many applications, this is considered sufficient protection. Scaling to prevent overflow is another solution, if as well the inital state values are never initialized to limit-cycle-producing values. The normal-form structure also reduces the chance of overflow.

Small-scale limit cycles

Small-scale limit cycles are caused by quantization. Consider the system



Note that when $\alpha z \quad z \quad \frac{\Delta_B}{}$, rounding will quantize the output to the current level (with zero input), so the output will remain at this level forever. Note that the maximum amplitude of this "small-scale limit cycle" is achieved when

$$\alpha z \quad z \quad \frac{\Delta_B}{} \qquad z \quad \frac{\Delta_B}{\alpha}$$

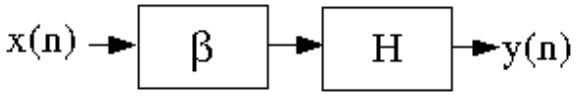
In a higher-order system, the small-scale limit cycles are oscillatory in nature. Any quantization scheme that never increases the magnitude of any quantized value prevents small-scale limit cycles.

Note: Two's-complement truncation does **not** do this; it increases the magnitude of negative numbers.

However, this introduces greater error and bias. Since the level of the limit cycles is proportional to Δ_B, they can be reduced by increasing the number of bits. Poles close to the unit circle increase the magnitude and likelihood of small-scale limit cycles.

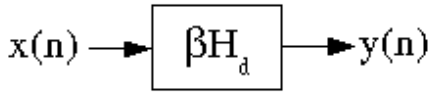
Scaling

Overflow is clearly a serious problem, since the errors it introduces are very large. As we shall see, it is also responsible for large-scale limit cycles, which cannot be tolerated. One way to prevent overflow, or to render it acceptably unlikely, is to **scale** the input to a filter such that overflow cannot (or is sufficiently unlikely to) occur.



In a fixed-point system, the range of the input signal is limited by the fractional fixed-point number representation to $|x[n]| \leq 1$. If we scale the input by multiplying it by a value β, $0 < \beta < 1$, then $|\beta x[n]| \leq \beta$.

Another option is to incorporate the scaling directly into the filter coefficients.



FIR Filter Scaling

What value of β is required so that the output of an FIR filter cannot overflow ($\forall n : (|y(n)| \leq 1)$, $\forall n : (|x(n)| \leq 1)$)?

$$|y(n)| = \left| \sum_{k=0}^{M-1} h(k)\beta x(n-k) \right| \leq \sum_{k=0}^{M-1} |h(k)|\,|\beta|\,|x(n-k)| \leq \beta \sum_{k=0}^{M-1} |h(k)| 1$$

$$\Downarrow$$

$$\beta < \sum_{k=0}^{M-1} |h(k)|$$

Alternatively, we can incorporate the scaling directly into the filter, and require that

$$\sum_{k=0}^{M-1} |h(k)| < 1$$

to prevent overflow.

IIR Filter Scaling

To prevent the output from overflowing in an IIR filter, the condition above still holds: ($M = \infty$)

$$|y(n)| < \sum_{k=0}^{\infty} |h(k)|$$

so an initial scaling factor $\beta < \frac{1}{\sum_{k=0}^{\infty} |h(k)|}$ can be used, or the filter itself can be scaled.

However, it is also necessary to prevent the **states** from overflowing, and to prevent overflow at any point in the signal flow graph where the arithmetic hardware would thereby produce errors. To prevent the states from overflowing, we determine the transfer function from the input to all states i, and scale the filter such that $\forall i : (\sum_{k=0}^{\infty} |h_i(k)| \leq 1)$

Although this method of scaling guarantees no overflows, it is often too conservative. Note that a worst-case signal is $x(n) = \text{sign}\,(h(-n))$; this input may be extremely unlikely. In the relatively common situation in which the input is expected to be mainly a single-frequency sinusoid of unknown frequency and amplitude less than 1, a scaling condition of

$$\forall w : (|H(w)| \leq 1)$$

is sufficient to guarantee no overflow. This scaling condition is often used. If there are several potential overflow locations i in the digital filter structure, the scaling conditions are

$$\forall i, w : (|H_i(w)| \leq 1)$$

where $H_i(w)$ is the frequency response from the input to location i in the filter.

Even this condition may be excessively conservative, for example if the input is more-or-less random, or if occasional overflow can be tolerated. In practice, experimentation and simulation are often the best ways to optimize the scaling factors in a given application.

For filters implemented in the cascade form, rather than scaling for the entire filter at the beginning, (which introduces lots of quantization of the input) the filter is usually scaled so that each stage is just prevented from overflowing. This is best in terms of reducing the quantization noise. The scaling factors are incorporated either into the previous or the next stage, whichever is most convenient.

Some heurisitc rules for grouping poles and zeros in a cascade implementation are:

1. Order the poles in terms of decreasing radius. Take the pole pair closest to the unit circle and group it with the zero pair closest to that pole pair (to minimize the gain in that section). Keep doing this with all remaining poles and zeros.
2. Order the section with those with highest gain ($\operatorname{argmax} |H_i(w)|$) in the middle, and those with lower gain on the ends.

Leland B. Jackson has an excellent intuitive discussion of finite-precision problems in digital filters. The book by Roberts and Mullis is one of the most thorough in terms of detail.

Introduction to Adaptive Filters

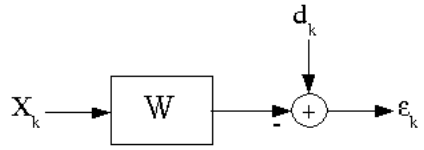
In many applications requiring filtering, the necessary frequency response may not be known beforehand, or it may vary with time. (Example; suppression of engine harmonics in a car stereo.) In such applications, an adaptive filter which can automatically design itself and which can track system variations in time is extremely useful. Adaptive filters are used extensively in a wide variety of applications, particularly in telecommunications.

Outline of adaptive filter material

1. **Wiener Filters** L optimal (FIR) filter design in a statistical context
2. **LMS algorithm** simplest and by-far-the-most-commonly-used adaptive filter algorithm
3. **Stability and performance of the LMS algorithm** When and how well it works
4. **Applications of adaptive filters** Overview of important applications
5. **Introduction to advanced adaptive filter algorithms** Techniques for special situations or faster convergence

Discrete-Time, Causal Wiener Filter

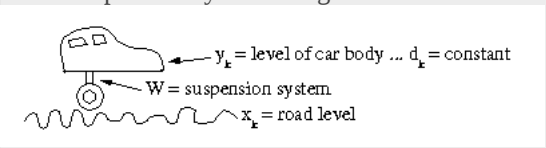
Stochastic L_2 optimal (least squares) FIR filter design problem: Given a wide-sense stationary (WSS) input signal x_k and desired signal d_k (WSS $\Leftrightarrow E[y_k] = E[y_{k+d}]$, $r_{yz}(l) = E[y_k z_{k+l}]$, $\forall k, l : (r_{yy}(0) < \infty)$)



The Wiener filter is the linear, time-invariant filter minimizing $E\left[\varepsilon^2\right]$, the variance of the error.

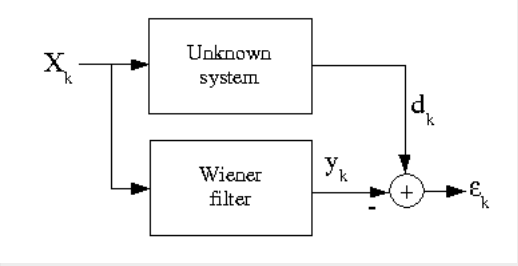
As posed, this problem seems slightly silly, since d_k is already available! However, this idea is useful in a wide cariety of applications.

Example:
active suspension system design



Note:optimal system may change with different road conditions or mass in car, so an **adaptive** system might be desirable.

Example:
System identification (radar, non-destructive testing, adaptive control systems)



Exercise:

Problem:

Usually one desires that the input signal x_k be "persistently exciting," which, among other things, implies non-zero energy in all frequency bands. Why is this desirable?

Determining the optimal length-N causal FIR Weiner filter

Note: for convenience, we will analyze only the causal, real-data case; extensions are straightforward.

$$y_k = \sum_{l=0}^{M-1} w_l x_{k-l}$$

$$\underset{w_l}{\operatorname{argmin}}\, E\left[\varepsilon^2\right] = E\left[(d_k - y_k)^2\right] = E\left[d_k - \sum_{l=0}^{M-1} w_l x_{k-l}\right]^2 = E\left[{d_k}^2\right] - 2\sum_{l=0}^{M-1} w_l E[d_k x_{k-l}] + \sum_{l=0}^{M-1}\sum_{m=0}^{M-1}(w_l$$

$$E\left[\varepsilon^2\right] = r_{\mathrm{dd}}(0) - 2\sum_{l=0}^{M-1} w_l r_{\mathrm{dx}}(l) + \sum_{l=0}^{M-1}\sum_{m=0}^{M-1} w_l w_m r_{\mathrm{xx}}(l-m)$$

where

$$r_{\mathrm{dd}}(0) = E\left[{d_k}^2\right]$$

$$r_{\mathrm{dx}}(l) = E[d_k X_{k-l}]$$

$$r_{\mathrm{xx}}(l-m) = E[x_k x_{k+l-m}]$$

This can be written in matrix form as

$$E\left[\varepsilon^2\right] = r_{\mathrm{dd}}(0) - 2\quad^T + \quad^T R$$

where

$$= \begin{matrix} r_{\mathrm{dx}}(0) \\ r_{\mathrm{dx}}(1) \\ \vdots \\ r_{\mathrm{dx}}(M-1) \end{matrix}$$

$$R = \begin{matrix} r_{\mathrm{xx}}(0) & r_{\mathrm{xx}}(1) & \dots & \dots & r_{\mathrm{xx}}(M-1) \\ r_{\mathrm{xx}}(1) & r_{\mathrm{xx}}(0) & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & r_{\mathrm{xx}}(0) & r_{\mathrm{xx}}(1) \\ r_{\mathrm{xx}}(M-1) & \dots & \dots & r_{\mathrm{xx}}(1) & r_{\mathrm{xx}}(0) \end{matrix}$$

To solve for the optimum filter, compute the gradient with respect to the top weights vector W

$$\nabla \doteq \begin{matrix} \frac{\partial \varepsilon^2}{\partial w_0} \\ \frac{\partial \varepsilon^2}{\partial w_1} \\ \vdots \\ \frac{\partial \varepsilon^2}{\partial w_{M-1}} \end{matrix}$$

$$\nabla = -(2 \quad) + 2R$$

(recall $\frac{\mathrm{d}}{\mathrm{d}}\left(\quad {}^{T}W\right) = \quad {}^{T}$, $\frac{\mathrm{d}}{\mathrm{d}}(\quad M \quad) = 2M$ for symmetric M) setting the gradient equal to zero ⇒

$$W_{\text{opt}}R = \quad \Rightarrow W_{\text{opt}} = R^{-1}$$

Since R is a correlation matrix, it must be non-negative definite, so this is a minimizer. For R positive definite, the minimizer is unique.

Practical Issues in Wiener Filter Implementation

The weiner-filter, $W_{\text{opt}} = R^{-1}\boldsymbol{P}$, is ideal for many applications. But several issues must be addressed to use it in practice.

Exercise:

Problem:

In practice one usually won't know exactly the statistics of x_k and d_k (i.e. R and $\boldsymbol{P}$) needed to compute the Weiner filter.

How do we surmount this problem?

Solution:

Estimate the statistics

$$r_{\text{xx}}(l) \simeq \frac{1}{N} \sum_{k=0}^{N-1} x_k x_{k+l}$$

$$r_{\text{xd}}(l) \simeq \frac{1}{N} \sum_{k=0}^{N-1} d_k x_{k-l}$$

then solve $W_{\text{opt}} = R^{-1} = P$

Exercise:

Problem:

In many applications, the statistics of x_k, d_k vary slowly with time.

How does one develop an **adaptive** system which tracks these changes over time to keep the system near optimal at all times?

Solution:

Use short-time windowed estiamtes of the correlation functions.

Note:

$$r_{\text{xx}}(l)^{k} = \frac{1}{N}\sum_{m\ 0}^{N-1} x_{k-m}x_{k-m-l}$$

$$r_{\text{dx}}(l)^{k} = \frac{1}{N}\sum_{m\ 0}^{N-1} x_{k-m-l}d_{k-m}$$

and $W_{\text{opt}}{}^{k} \simeq \quad R_{k}{}^{-1}P_{k}$

Exercise:

Problem: How can $r_{\text{xx}}^{k}(l)$ be computed efficiently?

Solution:

Recursively!

$$r_{\text{xx}}^{k}(l) = r_{\text{xx}}^{k-1}(l) + x_{k}x_{k-l} - x_{k-N}x_{k-N-l}$$

This is critically stable, so people usually do

$$(1-\alpha)\ \ r_{\text{xx}}{}^{k}(l) = \alpha r_{\text{xx}}^{k-1}(l) + x_{k}x_{k-l}$$

Exercise:

Problem: how does one choose N?

Tradeoffs

Larger $N \rightarrow$ more accurate estimates of the correlation values $\rightarrow$ better W_{opt}. However, larger N leads to slower adaptation.

Note: The success of adaptive systems depends on x, d being roughly stationary over at least N samples, $N > M$. That is, all adaptive filtering algorithms require that the underlying system varies slowly with respect to the sampling rate and the filter length (although they can tolerate occasional step discontinuities in the underlying system).

Computational Considerations

As presented here, an adaptive filter requires computing a matrix inverse at each sample. Actually, since the matrix R is Toeplitz, the linear system of equations can be sovled with $O\ M^2$ computations using Levinson's algorithm, where M is the filter length. However, in many applications this may be too expensive, especially since computing the filter output itself requires $O(M)$ computations. There are two main approaches to resolving the computation problem

1. Take advantage of the fact that R^{k+1} is only slightly changed from R^k to reduce the computation to $O(M)$; these algorithms are called Fast Recursive Least Squareds algorithms; all methods proposed so far have stability problems and are dangerous to use.
2. Find a different approach to solving the optimization problem that doesn't require explicit inversion of the correlation matrix.

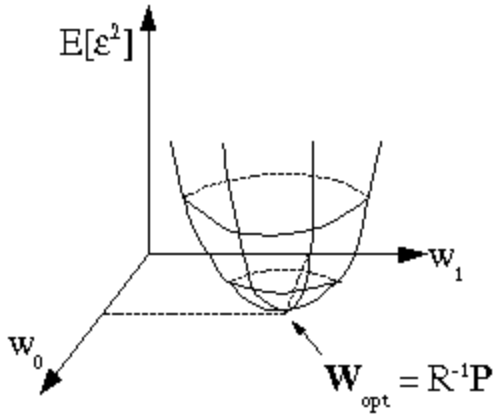
Note: Adaptive algorithms involving the correlation matrix are called **Recursive least Squares** (RLS) algorithms. Historically, they were developed after the LMS algorithm, which is the slimplest and most widely used approach $O(M)$. $O\ M^2$ RLS algorithms are used in applications requiring very fast adaptation.

Quadratic Minimization and Gradient Descent

Quadratic minimization problems

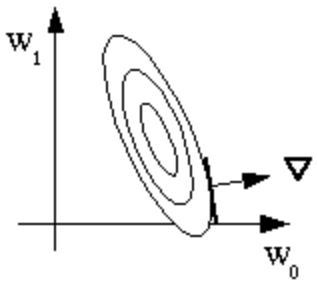
The least squares optimal filter design problem is quadratic in the filter coefficients:

If is positive definite, the error surface is a unimodal "bowl" in .



The problem is to find the bottom of the bowl. In an adaptive filter context, the shape and bottom of the bowl may drift slowly with time; hopefully slow enough that the adaptive algorithm can track it.

For a quadratic error surface, the bottom of the bowl can be found in one step by computing . Most modern nonlinear optimization methods (which are used, for example, to solve the optimal IIR filter design problem!) locally approximate a nonlinear function with a second-order (quadratic) Taylor series approximation and step to the bottom of this quadratic approximation on each iteration. However, an older and simpler appraoch to nonlinear optimaztion exists, based on **gradient descent**.
Contour plot of ε-squared



The idea is to iteratively find the minimizer by computing the gradient of the error function: ———. The gradient is a vector in pointing in the steepest uphill direction on the error surface at a given point , with having a magnitude proportional to the slope of the error surface in this steepest direction.

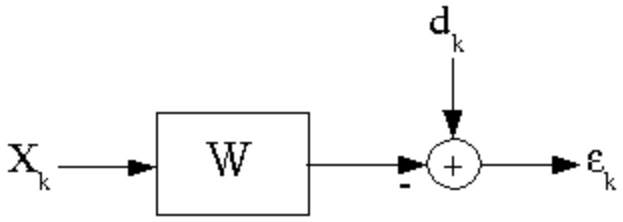
By updating the coefficient vector by taking a step **opposite** the gradient direction : , we go (locally) "downhill" in the steepest direction, which seems to be a sensible way to iteratively solve a nonlinear optimization problem. The performance obviously depends on ; if is too large, the iterations could bounce back and forth up out of the bowl. However, if is too small, it could take many iterations to approach the bottom. We will determine criteria for choosing later.

In summary, the gradient descent algorithm for solving the Weiner filter problem is:

The gradient descent idea is used in the LMS adaptive fitler algorithm. As presented, this alogrithm costs computations per iteration and doesn't appear very attractive, but LMS only requires computations and is stable, so it is very attractive when computation is an issue, even thought it converges more slowly then the RLS algorithms we have discussed so far.

The LMS Adaptive Filter Algorithm

Recall the Weiner filter problem



, jointly wide sense stationary

Find minimizing

The superscript denotes absolute time, and the subscript denotes time or a vector index.

the solution can be found by setting the gradient
Equation:

Alternatively, can be found iteratively using a gradient descent technique

In practice, we don't know and exactly, and in an adaptive context they may be slowly varying with time.

To find the (approximate) Wiener filter, some approximations are necessary. As always, the key is to make the **right** approximations!

Note: Approximate and : ⇒ RLS methods, as discussed last time.

Note: Approximate the gradient!

Note that itself is a very noisy approximation to . We can get a noisy approximation to the gradient by finding the gradient of ! Widrow and Hoff first published the LMS algorithm, based on this clever idea, in 1960.

This yields the LMS adaptive filter algorithm

Example:
The LMS Adaptive Filter Algorithm

1.
2.
3. (
)

The LMS algorithm is often called a **stochastic gradient** algorithm, since is a noisy gradient. This is **by far** the most commonly used adaptive filtering algorithm, because

1. it was the first
2. it is very simple
3. in practice it works well (except that sometimes it converges slowly)
4. it requires relatively litle computation
5. it updates the tap weights every sample, so it continually adapts the filter
6. it tracks slow changes in the signal statistics well

Computational Cost of LMS

To Compute ⇒				**= Total**
multiplies				
adds				

So the LMS algorithm is per sample. In fact, it is nicely balanced in that the filter computation and the adaptation require the same amount of computation.

Note that the parameter plays a very important role in the LMS algorithm. It can also be varied with time, but usually a constant ("convergence weight facor") is used, chosen after experimentation for a given application.

Tradeoffs

large : fast convergence, fast adaptivity

small : accurate → less misadjustment error, stability

First Order Convergence Analysis of the LMS Algorithm

Analysis of the LMS algorithm

It is important to analyze the LMS algorithm to determine under what conditions it is stable, whether or not it converges to the Wiener solution, to determine how quickly it converges, how much degredation is suffered due to the noisy gradient, etc. In particular, we need to know how to choose the parameter μ.

Mean of W

does W^k, $k \to \infty$ approach the Wiener solution? (since W^k is always somewhat random in the approximate gradient-based LMS algorithm, we ask whether the expected value of the filter coefficients converge to the Wiener solution)
Equation:

$$\begin{aligned} E\ W^{k+1} &= W^{k+1} \\ &= E\ W^k + 2\mu\varepsilon_k X^k \\ &= W^k + 2\mu E\ d_k X^k\ + 2\mu E\ -\quad W^{k^T} X^k\ \ X^k \\ &= W^k + 2\mu P + -\ \ 2\mu E\quad W^{k^T} X^k\ \ X^k \end{aligned}$$

Patently False Assumption

X^k and X^{k-i}, X^k and d^{k-i}, and d_k and d_{k-i} are statistically independent, $i \neq 0$. This assumption is obviously false, since X^{k-1} is the same as X^k except for shifting down the vector elements one place and adding one new sample. We make this assumption because otherwise it becomes extremely difficult to analyze the LMS algorithm. (First good analysis not making this assumption: Macchi and Eweda) Many simulations and much practical experience has shown that the results one obtains with analyses based on the patently false assumption above are quite accurate in most situations

With the independence assumption, W^k (which depends only on previous X^{k-i}, d^{k-i}) is statitically independent of X^k, and we can simplify $E \quad W^{k^T} X^k \quad X^k$

Now $\quad W^{k^T} X^k \quad X^k$ is a vector, and

Equation:

$$
\begin{aligned}
E \quad W^{k^T} X^k \quad X^k \quad &= \quad E \quad \begin{matrix} \vdots \\ \sum_{i=0}^{M-1} w_i^k x_{k-i} x_{k-j} \\ \vdots \end{matrix} \\
&= \quad \begin{matrix} \vdots \\ \sum_{i=0}^{M-1} E \ w_i^k x_{k-i} x_{k-j} \\ \vdots \\ \vdots \end{matrix} \\
&= \quad \begin{matrix} \vdots \\ \sum_{i=0}^{M-1} \quad w_i^k \ E[x_{k-i} x_{k-j}] \\ \vdots \\ \vdots \end{matrix} \\
&= \quad \begin{matrix} \vdots \\ \sum_{i=0}^{M-1} w_i^k r_{\mathrm{xx}}(i-j) \\ \vdots \end{matrix} \\
&= \quad RW^k
\end{aligned}
$$

where $R = E \ X^k X^{k^T}$ is the data correlation matrix.

Putting this back into our equation

Equation:

$$
\begin{aligned}
W^{k+1} \quad &= \quad W^k + 2\mu P + - \quad 2\mu R W^k \\
&= \quad I W^k + 2\mu P
\end{aligned}
$$

Now **if** $W^{k \to \infty}$ converges to a vector of finite magnitude ("convergence in the mean"), what does it converge to?

If W^k converges, then as $k \to \infty$, $W^{k+1} \simeq W^k$, and

$$W^\infty = IW^\infty + 2\mu P$$

$$2\mu RW^\infty = 2\mu P$$

$$RW^\infty = P$$

or

$$W_{\text{opt}} = R^{-1}P$$

the Wiener solution!

So the LMS algorithm, **if** it converges, gives filter coefficients which on average are the Wiener coefficients! This is, of course, a desirable result.

First-order stability

But does W^k converge, or under what conditions?

Let's rewrite the analysis in term of V^k, the "mean coefficient error vector"

$V^k = W^k - W_{\text{opt}}$, where W_{opt} is the Wiener filter

$$W^{k+1} = W^k - 2\mu RW^k + 2\mu P$$

$$W^{k+1} - W_{\text{opt}} = W^k - W_{\text{opt}} + - \quad 2\mu RW^k \quad + 2\mu RW_{\text{opt}} - 2\mu RW_{\text{opt}} + 2\mu P$$

$$V^{k+1} = V^k - 2\mu R V^k + -(2\mu R W_{\text{opt}}) + 2\mu P$$

Now $W_{\text{opt}} = R^{-1}$, so

$$V^{k+1} = V^k - 2\mu R V^k + - \ \ 2\mu R R^{-1} P \ \ + 2\mu P = (I - 2\mu R) V^k$$

We wish to know under what conditions $V^{k \to \infty} \to \bar{0}$?

Linear Algebra Fact

Since R is positive definite, real, and symmetric, all the eigenvalues are real and positive. Also, we can write R as $Q^{-1}\Lambda Q$, where Λ is a diagonal matrix with diagonal entries λ_i equal to the eigenvalues of R, and Q is a unitary matrix with rows equal to the eigenvectors corresponding to the eigenvalues of R.

Using this fact,

$$V^{k+1} = \ \ I - 2\mu \ \ Q^{-1}\Lambda Q \quad V^k$$

multiplying both sides through on the left by Q: we get

$$QV^{k+1} = (Q - 2\mu \Lambda Q) V^k = (1 - 2\mu\Lambda) Q V^k$$

Let $V \ \ = QV$:

$$V^{\ k+1} = (1 - 2\mu\Lambda) V^{\ k}$$

Note that $V \ \ $ is simply V in a rotated coordinate set in $\quad^m$, so convergence of V implies convergence of V.

Since $1 - 2\mu\Lambda$ is diagonal, all elements of $V \ \ $ evolve independently of each other. Convergence (stability) bolis down to whether all M of these scalar, first-order difference equations are stable, and thus $\to (0)$.

$$\forall i, i = [1, 2, \ldots, M] : \ \ V_i^{\ k+1} = (1 - 2\mu\lambda_i) V_i^{\ k}$$

These equations converge to zero if $|1 - 2\mu\lambda_i| < 1$, or $\forall i : (|\mu\lambda_i| < 1)$ μ and λ_i are positive, so we require $\forall i : \quad \mu < \frac{1}{\lambda_i}$ so for convergence in the mean of the LMS adaptive filter, we require
Equation:

$$\mu < \frac{1}{\lambda_{\max}}$$

This is an elegant theoretical result, but in practice, we may not know $\lambda_{\max}$, it may be time-varying, and we certainly won't want to compute it. However, another useful mathematical fact comes to the rescue...

$$\operatorname{tr}(R) = \sum_{i=1}^{M} r_{\mathrm{ii}} = \sum_{i=1}^{M} \lambda_i \geq \lambda_{\max}$$

Since the eigenvalues are all positive and real.

For a correlation matrix, $\forall i, i \in \{1, M\} : (r_{\mathrm{ii}} = r(0))$. So $\operatorname{tr}(R) = Mr(0) = ME[x_k x_k]$. We can easily estimate $r(0)$ with $O(1)$ computations/sample, so in practice we might require

$$\mu < \frac{1}{Mr(0)}$$

as a conservative bound, and perhaps adapt μ accordingly with time.

Rate of convergence

Each of the modes decays as

$$(1 - 2\mu\lambda_i)^k$$

Note: The **initial** rate of convergence is dominated by the fastest mode $1 - 2\mu\lambda_{\max}$. This is not surprising, since a dradient descent method goes "downhill" in the steepest direction

Note: The **final** rate of convergence is dominated by the slowest mode $1 - 2\mu\lambda_{\min}$. For small $\lambda_{\min}$, it can take a long time for LMS to converge.

Note that the convergence behavior depends on the data (via R). LMS converges relatively quickly for roughly equal eigenvalues. Unequal eigenvalues slow LMS down a lot.

Second-order Convergence Analysis of the LMS Algorithm and Misadjustment Error

Convergence of the mean (first-order analysis) is insufficient to guarantee desirable behavior of the LMS algorithm; the variance could still be infinite. It is important to show that the variance of the filter coefficients is finite, and to determine how close the average squared error is to the minimum possible error using an exact Wiener filter.

Equation:

$$\begin{aligned} E\left[{\varepsilon_k}^2\right] &= E\left[\left(d_k - {W^k}^T X^k\right)^2\right] \\ &= E\left[{d_k}^2 - 2d_k {X^k}^T W^k - {W^k}^T X^k {X^k}^T W^k\right] \\ &= r_{\mathrm{dd}}(0) - 2{W^k}^T P + {W^k}^T R W^k \end{aligned}$$

The minimum error is obtained using the Wiener filter

$$W_{\mathrm{opt}} = R^{-1}P$$

Equation:

$$\begin{aligned} {\varepsilon_{\min}}^2 &= E\left[\varepsilon^2\right] \\ &= \left(r_{\mathrm{dd}}(0) - 2P^T R^{-1} P + P^T R^{-1} R R^{-1} P\right) \\ &= r_{\mathrm{dd}}(0) - P^T R^{-1} P \end{aligned}$$

To analyze the average error in LMS, write [link] in terms of $V' = Q[W - W_{\mathrm{opt}}]$, where $Q^{-1}\Lambda Q = R$

Equation:

$$\begin{aligned} E\left[{\varepsilon_k}^2\right] &= r_{\mathrm{dd}}(0) - 2{W^k}^T P + {W^k}^T R W^k + \left(-\left({W^k}^T R W_{\mathrm{opt}}\right)\right) - {W_{\mathrm{opt}}}^T R W^k + {W_{\mathrm{opt}}}^T R W_{\mathrm{opt}} + {W^k}^T R \\ &= r_{\mathrm{dd}}(0) + {V^k}^T R V^k - P^T R^{-1} P \\ &= {\varepsilon_{\min}}^2 + {V^k}^T R V^k \\ &= {\varepsilon_{\min}}^2 + {V^k}^T Q^{-1} Q R Q^{-1} Q V^k \\ &= {\varepsilon_{\min}}^2 + {V'^k}^T \Lambda V'^k \end{aligned}$$

$$E\left[{\varepsilon_k}^2\right] = {\varepsilon_{\min}}^2 + \sum_{j=0}^{N-1} \lambda_j E\left[{v_j'^k}^2\right]$$

So we need to know $E\left[{v_j'^k}^2\right]$, which are the diagonal elements of the covariance matrix of V'^k, or $E\left[V'^k {V'^k}^T\right]$.

From the LMS update equation

$$W^{k+1} = W^k + 2\mu\varepsilon_k X^k$$

we get

$$V'^{k+1} = W'^k + 2\mu\varepsilon_k Q X^k$$

Equation:

$$\begin{aligned}\mathcal{V}^{k+1} &= E\left[V'^{k+1}V'^{k+1^T}\right] \\ &= E\left[4\mu^2{\varepsilon_k}^2 QX^k X^{k^T}Q^T\right] \\ &= \mathcal{V}^k + 2\mu\left(\varepsilon_k QX^k V'^{k^T}\right) + 2\mu\left(\varepsilon_k V'^k X^{k^T}Q^T\right) + 4\mu^2 E\left[{\varepsilon_k}^2 QX^k X^{k^T}Q^T\right]\end{aligned}$$

Note that

$$\varepsilon_k = d_k - W^{k^T}X^k = d_k - {W_{\text{opt}}}^T - V'^{k^T}QX^k$$

so
Equation:

$$\begin{aligned}E\left[\varepsilon_k QX^k V'^{k^T}\right] &= E\left[d_k QX^k V'^{k^T} - {W_{\text{opt}}}^T X^k QX^k V'^{k^T} - V'^{k^T}QX^k V'^{k^T}\right] \\ &= 0 + 0 - \left(QX^k X^{k^T}Q^T V'^k V'^{k^T}\right) \\ &= -\left(QE\left[X^k X^{k^T}\right]Q^T E\left[V'^k V'^{k^T}\right]\right) \\ &= -\left(\Lambda\mathcal{V}^k\right)\end{aligned}$$

Note that the Patently False independence Assumption was invoked here.

To analyze $E\left[{\varepsilon_k}^2 QX^k X^{k^T}Q^T\right]$, we make yet another obviously false assumptioon that ${\varepsilon_k}^2$ and X^k are statistically independent. This is obviously false, since $\varepsilon_k = d_k - W^{k^T}X^k$. Otherwise, we get 4th-order terms in X in the product. These can be dealt with, at the expense of a more complicated analysis, if a particular type of distribution (such as Gaussian) is assumed. See, for example Gardner. A questionable justification for this assumption is that as $W^k \simeq W_{\text{opt}}$, W^k becomes uncorrelated with X^k (if we invoke the original independence assumption), which tends to randomize the error signal relative to X^k. With this assumption,

$$E\left[{\varepsilon_k}^2 QX^k X^{k^T}Q^T\right] = E\left[{\varepsilon_k}^2\right]E\left[QX^k X^{k^T}Q^T\right] = E\left[{\varepsilon_k}^2\right]\Lambda$$

Now

$${\varepsilon_k}^2 = {\varepsilon_{\min}}^2 + V'^{k^T}\Lambda V'^k$$

so
Equation:

$$\begin{aligned}E\left[{\varepsilon_k}^2\right] &= {\varepsilon_{\min}}^2 + E\left[\sum_j \lambda_j {V'_j}^{k^2}\right] \\ &= {\varepsilon_{\min}}^2 + \sum_j \lambda_j \mathcal{V}_{\text{jj}}^k\end{aligned}$$

Thus, [link] becomes
Equation:

$$\mathcal{V}^{k+1} = I\mathcal{V}^k + 4\mu^2\sum_j \lambda_j \mathcal{V}_{\text{jj}}^k \Lambda + 4\mu^2 {\varepsilon_{\min}}^2 \Lambda$$

Now **if** this system is stable and converges, it converges to $\mathcal{V}^\infty = \mathcal{V}^{\infty+1}$

$$\Rightarrow \left(4\mu\Lambda\mathscr{V}^{\infty} = 4\mu^2 \left(\sum_j \lambda_j \mathscr{V}_{\mathrm{jj}} + {\varepsilon_{\min}}^2 \right) \Lambda \right)$$

$$\Rightarrow \left(\mathscr{V}^{\infty} = \mu \left(\sum_j \lambda_j \mathscr{V}_{\mathrm{jj}} + {\varepsilon_{\min}}^2 \right) I \right)$$

So it is a diagonal matrix with all elements on the diagonal equal:

Then

$$\mathscr{V}_{\mathrm{ii}}^{\infty} = \mu \left(\mathscr{V}_{\mathrm{ii}}^{\infty} \sum_j \lambda_j + {\varepsilon_{\min}}^2 \right)$$

$$\mathscr{V}_{\mathrm{ii}}^{\infty} \left(1 - \mu \sum_j \lambda_j \right) = \mu {\varepsilon_{\min}}^2$$

$$\mathscr{V}_{\mathrm{ii}}^{\infty} = \frac{\mu {\varepsilon_{\min}}^2}{1 - \mu \sum_j \lambda_j}$$

Thus the error in the LMS adaptive filter after convergence is
Equation:

$$\begin{aligned} E\left[{\varepsilon_{\infty}}^2\right] &= {\varepsilon_{\min}}^2 + E\left[V'^{\infty} \lambda V'^{\infty}\right] \\ &= {\varepsilon_{\min}}^2 + \frac{\mu {\varepsilon_{\min}}^2 \sum_j \lambda_j}{1 - \mu \sum_j \lambda_j} \\ &= {\varepsilon_{\min}}^2 \frac{1}{1 - \mu \sum_j \lambda_j} \\ &= {\varepsilon_{\min}}^2 \frac{1}{1 - \mu \operatorname{tr}(R)} \\ &= {\varepsilon_{\min}}^2 \frac{1}{1 - \mu r_{\mathrm{xx}}(0)\mathrm{N}} \end{aligned}$$

Equation:

$$E\left[{\varepsilon_{\infty}}^2\right] = {\varepsilon_{\min}}^2 \frac{1}{1 - \mu \mathrm{N} {\sigma_x}^2}$$

$1 - \mu \mathrm{N} {\sigma_x}^2$ is called the **misadjustment factor**. Oftern, one chooses μ to select a desired misadjustment factor, such as an error 10% higher than the Wiener filter error.

2nd-Order Convergence (Stability)

To determine the range for μ for which [link] converges, we must determine the μ for which the matrix difference equation converges.

$$\mathscr{V}^{k+1} = I\mathscr{V}^{k} + 4\mu^2 \sum_j \lambda_j \mathscr{V}_{\mathrm{jj}}^{k} \Lambda + 4\mu^2 {\varepsilon_{\min}}^2 \Lambda$$

The off-diagonal elements each evolve independently according to $\mathscr{V}_{\mathrm{ij}}^{k+1} = 1 - 4\mu\lambda_i \mathscr{V}_{\mathrm{ij}}^{k}$ These terms will decay to zero if $\forall i : (4\mu\lambda_i < 2)$, or $\mu < \frac{1}{2\lambda_{\max}}$

The diagonal terms evolve according to

$$\mathcal{V}_{\mathrm{ii}}^{k+1} = 1\mathcal{V}_{\mathrm{ii}}^{k} + 4\mu^2\lambda_i \sum_j \lambda_j \mathcal{V}_{\mathrm{jj}}^{k} + 4\mu^2 {\varepsilon_{\min}}^2 \lambda_i$$

For the homoegeneous equation

$$\mathcal{V}_{\mathrm{ii}}^{k+1} = 1\mathcal{V}_{\mathrm{ii}}^{k} + 4\mu^2\lambda_i \sum_j \lambda_j \mathcal{V}_{\mathrm{jj}}^{k}$$

for $1 - 4\mu\lambda_i$ positive,
Equation:

$$\mathcal{V}_{\mathrm{ii}}^{k+1} \leq 1\mathcal{V}_{\mathrm{iimax}}^{k} + 4\mu^2\lambda_i \sum_j \lambda_j \mathcal{V}_{\mathrm{jjmax}}^{k} = \left(1 - 4\mu\lambda_i + 4\mu^2\lambda_i \sum_j \lambda_j\right) \mathcal{V}_{\mathrm{jjmax}}^{k}$$

$\mathcal{V}_{\mathrm{ii}}^{k+1}$ will be strictly less than $\mathcal{V}_{\mathrm{jjmax}}^{k}$ for

$$1 - 4\mu\lambda_i + 4\mu^2\lambda_i \sum_j \lambda_j < 1$$

or

$$4\mu^2\lambda_i \sum_j \lambda_j < 4\mu\lambda_i$$

or
Equation:

$$\begin{aligned} \mu < \frac{1}{\sum_j \lambda_j} &= \frac{1}{\mathrm{tr}\,(R)} \\ &= \frac{1}{N r_{\mathrm{xx}}(0)} \\ &= \frac{1}{N {\sigma_x}^2} \end{aligned}$$

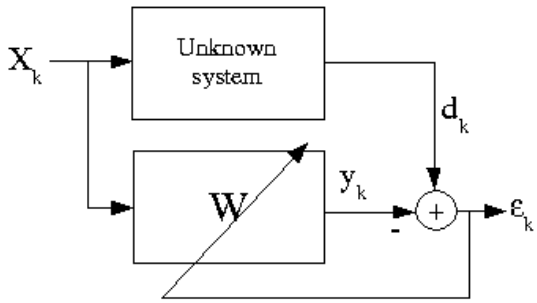
This is a more rigorous bound than the first-order bounds. Ofter engineers choose μ a few times smaller than this, since more rigorous analyses yield a slightly smaller bound. $\mu = \frac{\mu}{3N{\sigma_x}^2}$ is derived in some analyses assuming Gaussian x_k, d_k.

Applications of Adaptive Filters

Adaptive filters can be configured in a surprisingly large variety of ways for application in a large number of applications. The same LMS algorithm can be used to adapt the filter coefficients in most of these configurations; only the "plumbing" changes.

Adaptive System Identification

Note: To approximate an unknown system (or the behavior of that system) as closely as possible



The optimal solution is

Suppose the unknown system is a causal, linear time-invariant filter:

Now
Equation:

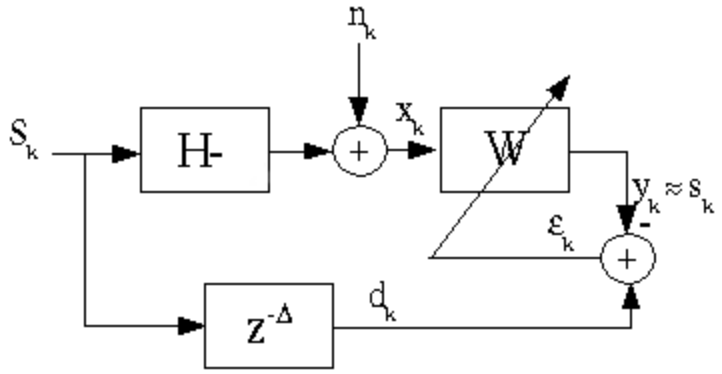
If the adaptive filter is a length- FIR filter ( ), this reduces to

and

FIR adaptive system identification thus converges in the mean to the corresponding samples of the impulse response of the unknown system.

Adaptive Equalization

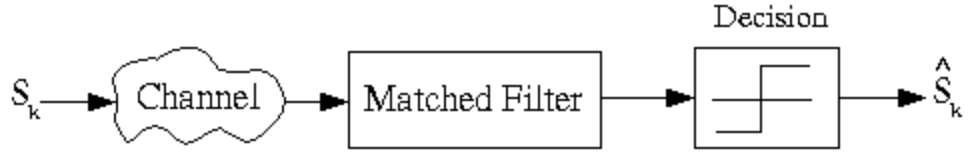
Note: Design an approximate inverse filter to cancel out as much distortion as possible.



In principle, $WH \simeq z^{-\Delta}$, or $W \simeq \frac{z^{-\Delta}}{H}$, so that the overall response of the top path is approximately $\delta(n - \Delta)$. However, limitations on the form of W (FIR) and the presence of noise cause the equalization to be imperfect.

Important Application

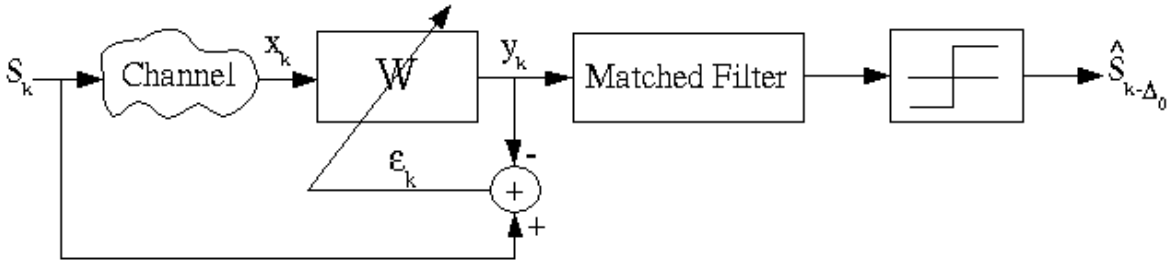
Channel equalization in a digital communication system.



If the channel distorts the pulse shape, the matched filter will no longer be matched, intersymbol interference may increase, and the system

performance will degrade.

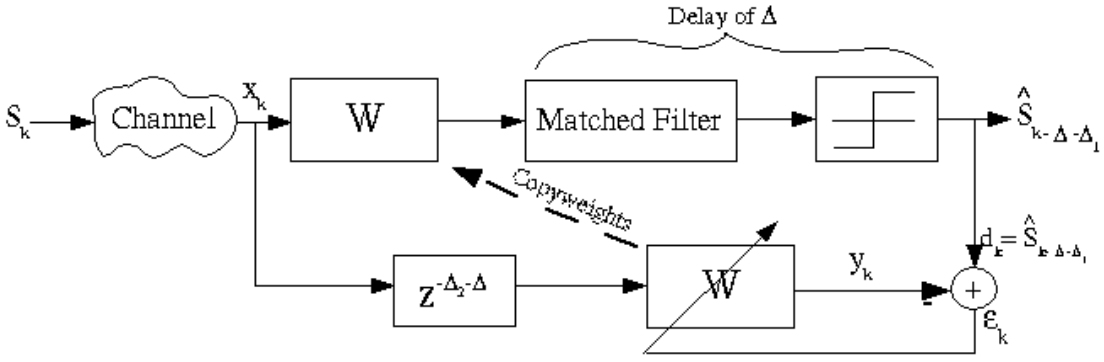
An adaptive filter is often inserted in front of the matched filter to compensate for the channel.



This is, of course, unrealizable, since we do not have access to the original transmitted signal, s_k.

There are two common solutions to this problem:

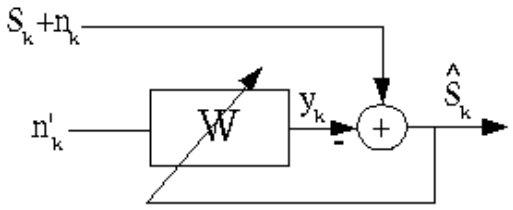
1. Periodically broadcast a known **training signal**. The adaptation is switched on only when the training signal is being broadcast and thus s_k is known.
2. Decision-directed feedback: If the overall system is working well, then the output $\hat{s}_{k-\Delta}$ should almost always equal $s_{k-\Delta}$. We can thus use our received digital communication signal as the desired signal, since it has been cleaned of noise (we hope) by the nonlinear threshold device!
   Decision-directed equalizer



As long as the error rate in $\hat{s}_k$ is not too high (say 75%), this method works. Otherwise, d_k is so inaccurate that the adaptive filter can never find the Wiener solution. This method is widely used in the telephone system and other digital communication networks.

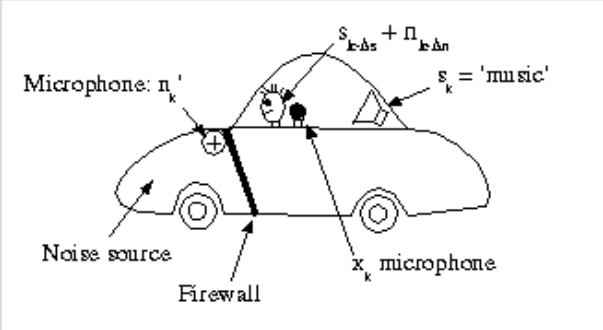
Adaptive Interference (Noise) Cancellation

Note: Automatically eliminate unwanted interference in a signal.

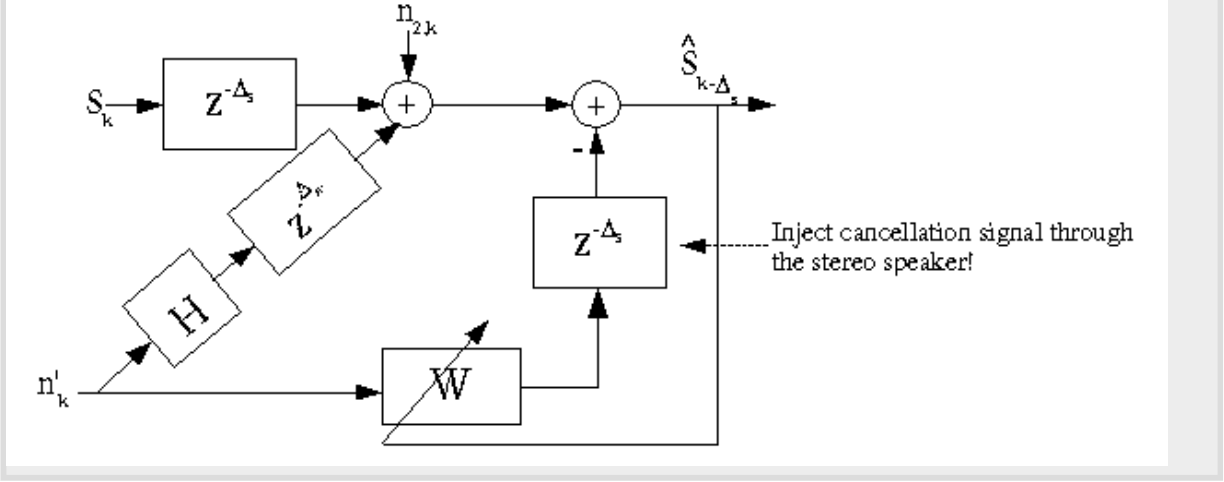


The object is to subtract out as much of the noise as possible.

Example:
Engine noise cancellation in automobiles



The firewall attenuates and filters the noise reaching the listener's ear, so it is not the same as . There is also a delay due to acoustic propagation in the air. For maximal cancellation, an adaptive filter is thus needed to make as similar as possible to the delayed .

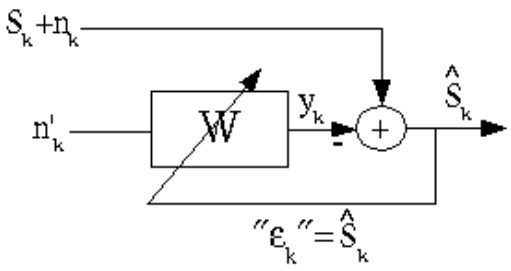


Exercise:

Problem:

What conditions must we impose upon the microphone locations for this to work? (Think causality and physics!)

Analysis of the interference cancellor

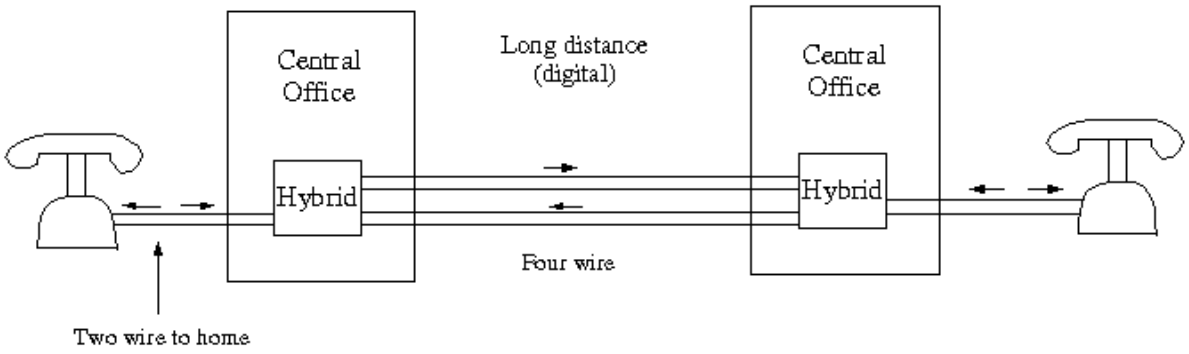


We assume , , and are zero-mean signals, and that is independent of and . Then

Since the input signal has no information about in it, minimizing can only affect the second term, which is the standard Wiener filtering problem, with solution

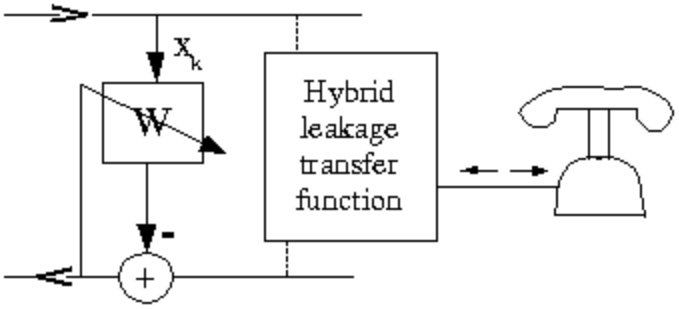
Adaptive Echo Cancellation

An adaptive echo canceller is a specific type of adaptive interference canceller that removes echos. Adaptive echo cancellers are found in all modern telephone systems.



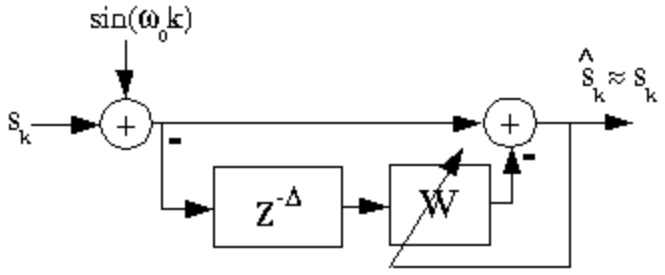
The hybrid is supposed to split the opposite-going waves, but typically achieves only about 15dB of suppression. This signal will eventually reach the other end and be coupled back, with a long delay, to the original source, which gives a very annoying echo.

Echo canceller



Because the input to the adaptive echo canceller contains only the signal from the far end that will echo off of the hybrid, it cancels the echo while passing the near-end signal as desired.

Narrowband interference canceller



A sinusoid is predictable samples ahead, whereas may not be, so the sinusoid can be cancelled using the adaptive system in the Figure. This is another special case of the adaptive interference canceller in which the noise reference input is a delayed version of the primary (signal plus noise) input. Note that must be large enough so that and are uncorrelated, or some of the signal will be cancelled as well!

Exercise:

Problem:

How would you construct an "adaptive line enhancer" that preserves the sinusoids but cancels the uncorrelated noise?

Other Applications

- Adaptive array processing
- Adaptive control
- etc...

Beyond LMS: an overview of other adaptive filter algorithms

RLS algorithms

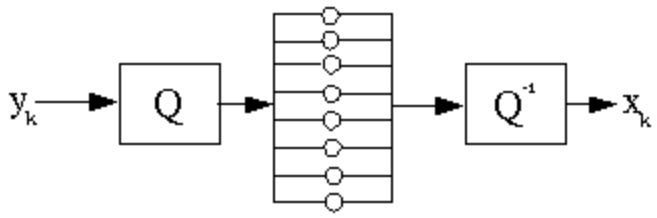
FIR adaptive filter algorithms with faster convergence. Since the Wiener solution can be obtained on one step by computing $W_{\text{opt}} = R^{-1}P$, most RLS algorithms attept to estimate R^{-1} and P and compute W_{opt} from these.

There are a number of $O\left(N^2\right)$ algorithms which are stable and converge quickly. A number of $O(N)$ algorithms have been proposed, but these are all unstable except for the lattice filter method. This is described to some extent in the text. The adaptive lattice filter converges quickly and is stable, but reportedly has a very high noise floor.

Many of these approaches can be thought of as attempting to "orthogonalize" R, or to rotate the data or filter coefficients to a domain where R is diagonal, then doing LMS in each dimension **separately**, so that a fast-converging step size can be chosen in all directions.

Frequency-domain methods

Frequency-domain methods implicitly attempt to do this:



If QRQ^{-1} is a diagonal matrix, this yields a fast algorithm. If Q is chosen as an FFT matrix, each channel becomes a different frequency bin. Since R is Toeplitz and not a circulant, the FFT matrix will not exactly diagonalize R, but in many cases it comes very close and frequency domain methods

converge very quickly. However, for some R they perform no better than LMS. By using an FFT, the transformation Q becomes inexpensive $O(N \log N)$. If one only updates on a block-by-block basis (once per N samples), the frequency domain methods only cost $O(\log N)$ computations per sample. which can be important for some applications with large N. (Say 16,000,000)

Adaptive IIR filters

Adaptive IIR filters are attractive for the same reasons that IIR filters are attractive: many fewer coefficients may be needed to achieve the desired performance in some applications. However, it is more difficult to develop stable IIR algorithms, they can converge **very** slowly, and they are susceptible to local minima. Nonetheless, adaptive IIR algorithms are used in some applications (such as low frequency noise cancellation) in which the need for IIR-type responses is great. In some cases, the exact algorithm used by a company is a tightly guarded trade secret.

Most adaptive IIR algorithms minimize the **prediction** error, to linearize the estimation problem, as in deterministic or block linear prediction.

Thus the coefficient vector is

and the "signal" vector is

The error is

An LMS algorithm can be derived using the approximation or

Now

Note that these are difference equations in ——, ——: call them ——, ——, then , and the IIR LMS algorithm becomes

and finally

where the may be **different** for the different IIR coefficients. Stability and convergence rate depends on these choices, of course. There are a number of variations on this algorithm.

Due to the slow convergence and the difficulties in tweaking the algorithm parameters to ensure stability, IIR algorithms are used only if there is an overriding need for an IIR-type filter.

The Constant-Modulus Algorithm and the Property-Restoral Principle

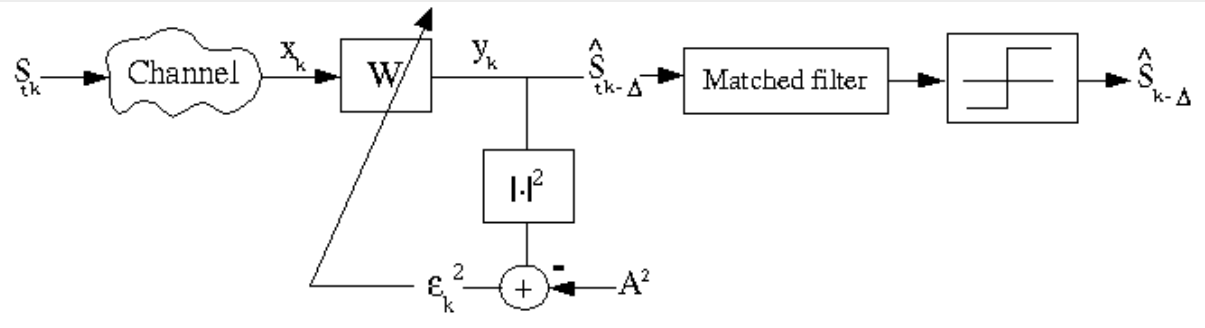
The adaptive filter configurations that we have examined so far require a "desired signal" d_k. There are many clever ways to obtain such a signal, but in some potential applications a desired signal is simply not available. However, a "property-restoral algorithm" can sometimes circumvent this problem.

If the uncorrupted signal has special properties that are characteristic of the signal and not of the distortion or interference, an algorithm can be constructed which attempts to cause the output of the adaptive filter to exhibit that property. Hopefully, the adapting filter will restore that property by removing the distortion or interference!

Example:
the Constant-Modulus Algorithm (CMA)
Certain communication modulation schemes, such as PSK and FSK, transmit a sinusoid of a constant analytic magnitude. Only the frequency or phase change with time. The constant modulus algorithm tries to drive the output signal to one having a constant amplitude:

$$\varepsilon_k = (|y_k|)^2 - A^2$$

One can derive an LMS (or other) algorithm that seeks a Wiener filter minimizing this error. In practice, this works very well for equalization of PSK and FSK communication channels.



CMA is simpler than decision-directed feedback, and can work for high initial error rates!

This property-restoral idea can be used in any context in which a property-related error can be defined.

Complex LMS

LMS for complex data and coefficients (such as quadrature communication systems) takes the form

$$y_k = W_k^H X_k$$

$$e_k = d_k - y_k$$

$$W_{k+1} = W_k + 2\mu e_k X_k$$

It is derived in exactly the same way as LMS, using the following complex vector differentiation formulas

$$\frac{\mathrm{d}}{\mathrm{d}} P^H = 0$$

$$\frac{\mathrm{d}}{\mathrm{d}} W^H = 2$$

$$\frac{\mathrm{d}}{\mathrm{d}} W^H R = 2R$$

or by differentiating with respect to the real and imaginary parts separately and recombining the results.

Normalized LMS

In "normalized" LMS, the gradient step factor μ is normalized by the energy of the data vector:

$$\mu_{\text{NLMS}} = \frac{\alpha}{X_k^H X_k + \sigma}$$

where α is usually $\frac{1}{2}$ and σ is a very small number introduced to prevent division by zero if $X_k^H X_k$ is very small.

$$W_{k+1} = W_k + \frac{1}{X^H} e_k X_k$$

The normalization has several interpretations

1. corresponds to the 2nd-order convergence bound
2. makes the algorithm independent of signal scalings
3. adjusts W_{k+1} to give zero error with current input: $W_{k+1} X_k = d_k$
4. minimizes mean effort at time $k + 1$

NLMS usually converges **much** more quickly than LMS at very little extra cost; NLMS is **very** commonly used. In some applications, normalization is so universal that "we use the LMS algorithm" implies normalization as well.

Summary of Adaptive Filtering Methods

1. **LMS** remains the simplest and best algorithm when slow convergence is not a serious issue (typically used) $O\ N$
2. **NLMS** simple extension of the LMS with much faster convergence in many cases (very commonly used) $O\ N$
3. **Frequency-domain methods** offer computational savings ($O\qquad N\ $) for long filters and usually offer faster convergence, too (sometimes used; very commonly used when there are already FFTs in the system)
4. **Lattice methods** are stable and converge quickly, but cost substantially more than LMS and have higher residual EMSE than many methods (very occasionally used) $O\ N$
5. **RLS** algorithms that converge quickly and are stable exist. However, they are considerably more expensive than LMS. (almost never used) $O\ N$
6. **Block RLS** (least squares) methods exist and can be pretty efficient in some cases. (occasionally used) $O\qquad N\ , O\ N\ , O\ N$
7. **IIR** methods are difficult to implement successfully and pose certain difficulties, but are sometimes used in some applications, for example noise cancellation of low frequency noise (very occasionally used)
8. **CMA** very useful when applicable (blind equalization); CMA is **the** method for blind equalizer initialization (commonly used in a few specific equalization applications) $O\ N$

Note: In general, getting adaptive filters to work well in an application is much more challenging than, say, FFTs or IIR filters; they generally require lots of tweaking!

Overview of Multirate Signal Processing

Digital transformation of the sampling rate of signals, or signal processing with different sampling rates in the system.

Applications

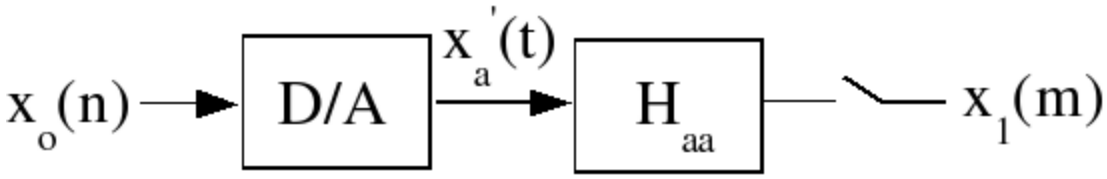
1. **Sampling-rate conversion** CD to DAT format change, for example.
2. **Improved D/A, A/D conversion** oversampling converters; which reduce performance requirements on anti-aliasing or reconstruction filters
3. **FDM channel modulation and processing** bandwidth of individual channels is much less than the overall bandwidth
4. **Subband coding of speech and images** Eyes and ears are not as sensitive to errors in higher frequency bands, so many coding schemes split signals into different frequency bands and quantize higher-frequency bands with much less precision.

Outline of Multirate DSP material

1. General Rate-changing System
2. Integer-factor Interpolation and Decimation and Rational-factor Rate Changing
3. Efficient Multirate Filter Structures
4. Optimal Filter Design for Multirate Systems
5. Multi-stage Multirate Systems
6. Oversampling D/As
7. Perfect-Reconstruction Filter Banks and Quadrature Mirror Filters

General Rate-Changing Procedure

This procedure is motivated by an analog-based method: one conceptually simple method to change the sampling rate is to simply convert a digital signal to an analog signal and resample it! ([link])



$$H_{\text{aa}}(\Omega) = \begin{array}{l} 1 \text{ if } |\Omega| < \frac{\pi}{T_1} \\ 0 \text{ otherwise} \end{array}$$

$$h_{\text{aa}}(t) = \frac{\sin \frac{\pi}{T_1} t}{\frac{\pi}{T_1} t}$$

Recall the ideal D/A:
Equation:

$$x'_a(t) = \sum_{n=-\infty}^{\infty} x_0(n) \frac{\sin \frac{\pi(t-nT_0)}{T_0}}{\frac{\pi(t-nT_0)}{T_0}}$$

The problems with this scheme are:

1. A/D, D/A, filters cost money
2. imperfections in these devices introduce errors

Digital implementation of rate-changing according to this formula has three problems:

1. Infinite sum: The solution is to truncate. Consider sinc $(t) \simeq 0$ for $t < t_1, t > t_2$: Then $mT_1 - nT_0 < t_1$ and $mT_1 - nT_0 > t_2$ which implies

$$N_1 = \frac{mT_1 - t_2}{T_0}$$

$$N_2 = \frac{mT_1 - t_1}{T_0}$$

$$x_1(m) = \sum_{n=N_1}^{N_2} x_0(n)\ \mathrm{sinc}_{T'}\ (mT_1 - nT_0)$$

> **Note:** This is essentially lowpass filter design using a boxcar window: other finite-length filter design methods could be used for this.

2. Lack of causality: The solution is to delay by $\max\{|N|\}$ samples. The mathematics of the analog portions of this system can be implemented digitally.
   Equation:

$$\begin{aligned} x_1(m) &= h_{\mathrm{aa}}(t) * x'_a(t)|_{t=mT_1} \\ &= \int_{-\infty}^{\infty} \sum_{n=-\infty}^{\infty} x_0(n) \frac{\sin \frac{\pi\ mT_1 - \tau - nT_0}{T_0}}{\frac{\pi\ mT_1 - \tau - nT_0}{T_0}} \frac{\sin \frac{\pi\tau}{T_1}}{\frac{\pi\tau}{T_1}}\ \mathrm{d}\,\tau \end{aligned}$$

   Equation:

$$\begin{aligned} x_1(m) &= \sum_{n=-\infty}^{\infty} x_0(n) \frac{\sin \frac{\pi}{T'}(mT_1 - nT_0)}{\frac{\pi}{T'}(mT_1 - nT_0)} \quad T' = \max\{T_0, T_1\} \\ &= \sum_{n=-\infty}^{\infty} x_0(n)\ \mathrm{sinc}_{T'}\ (mT_1 - nT_0) \end{aligned}$$

   So we have an all-digital formula for **exact** digital-to-digital rate changing!
3. Cost of computing $\mathrm{sinc}_{T'}\ (mT_1 - nT_0)$: The solution is to precompute the table of $\mathrm{sinc}\ (t)$ values. However, if $\frac{T_1}{T_0}$ is not a rational fraction, an infinite number of samples will be needed, so some approximation will have to be tolerated.

Note: Rate transformation of any rate to any other rate can be accomplished digitally with arbitrary precision (if some delay is acceptable). This method is used in practice in many cases. We will examine a number of special cases and computational improvements, but in some sense everything that follows are details; the above idea is the central idea in multirate signal processing.

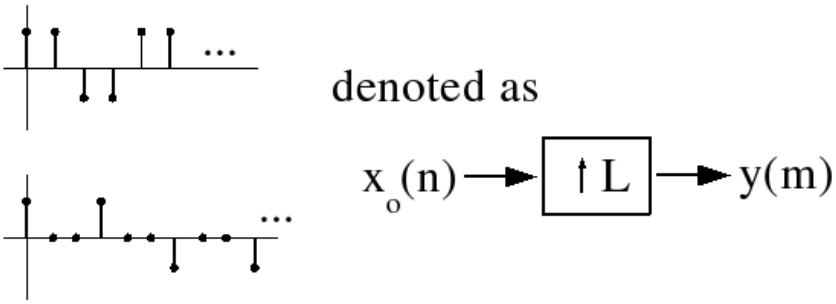
Useful references for the traditional material (everything except PRFBs) are Crochiere and Rabiner, 1981 and Crochiere and Rabiner, 1983. A more recent tutorial is Vaidyanathan; see also Rioul and Vetterli. References to most of the original papers can be found in these tutorials.

Interpolation, Decimation, and Rate Changing by Integer Fractions

Interpolation: by an integer factor L

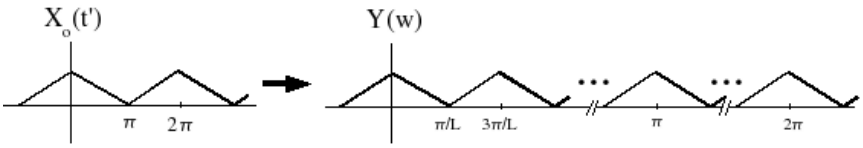
Interpolation means increasing the sampling rate, or filling in in-between samples. Equivalent to sampling a bandlimited analog signal times faster. For the ideal interpolator,
Equation:

We wish to accomplish this digitally. Consider [link] and [link].
Equation:



The DTFT of is
Equation:

Since is periodic with a period of , is periodic with a period of — (see [link]).



By inserting zero samples between the samples of , we obtain a signal with a scaled frequency response that simply replicates times over a interval!

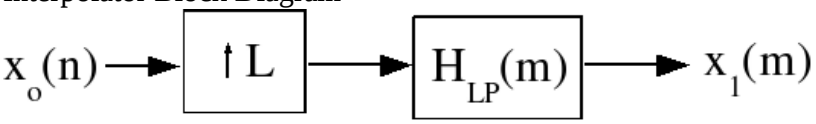
Obviously, the desired can be obtained simply by lowpass filtering to remove the replicas.

Equation:

Given

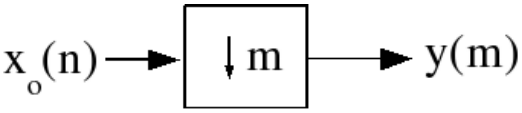
In practice, a finite-length lowpass filter is designed using any of the methods studied so far ([link]).

Interpolator Block Diagram

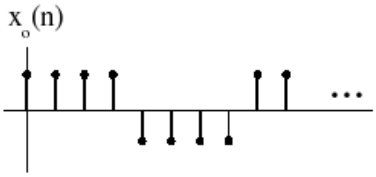


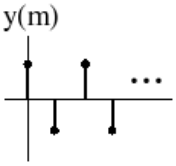
Decimation: sampling rate reduction (by an integer factor M)

Let ([link])



That is, keep only every th sample ([link])

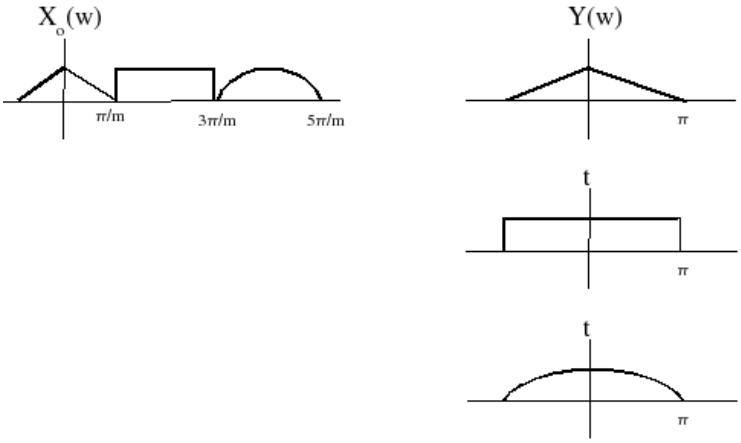




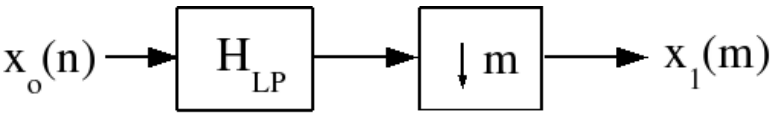
In frequency (DTFT):
Equation:

Now — for as shown in homework #1 , where is the DFT of one period of the periodic sequence. In this case, for and — .
Equation:

so — — i.e., we get **digital aliasing**.([link])

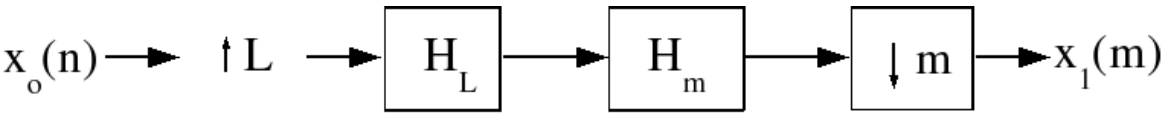


Usually, we prefer not to have aliasing, so the downsampler is preceded by a lowpass filter to remove all frequency components above — ([link]).



Rate-Changing by a Rational Fraction L/M

This is easily accomplished by interpolating by a factor of , then decimating by a factor of ([link]).



The two lowpass filters can be combined into one LP filter with the lower cutoff,

———

———

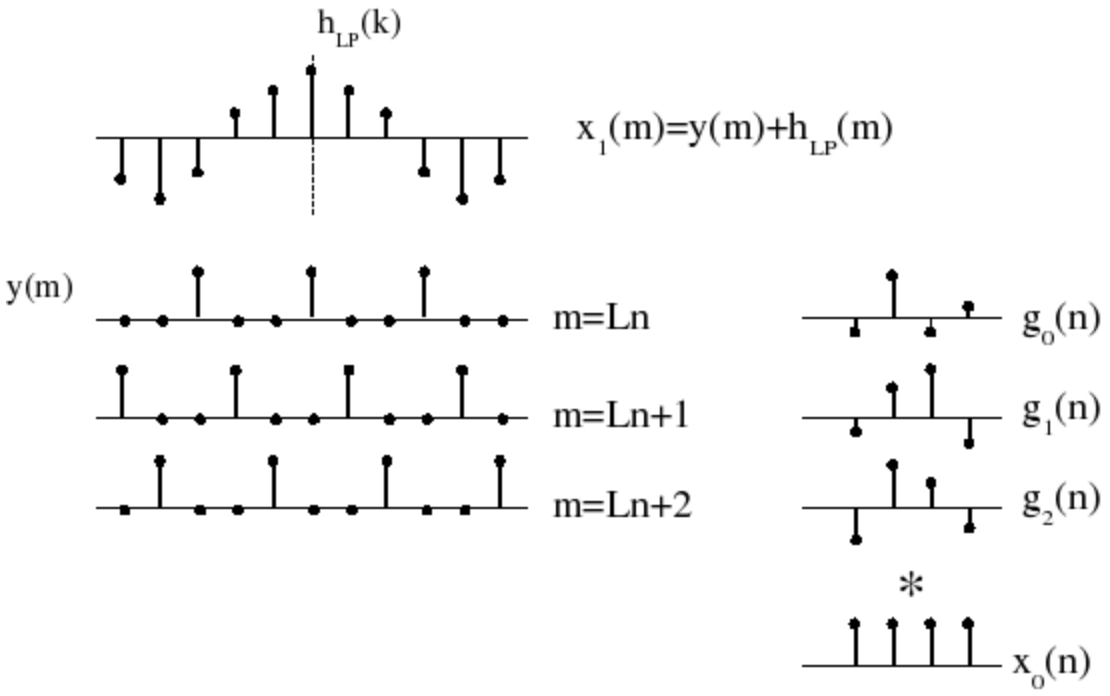
Obviously, the computational complexity and simplicity of implementation will depend on —: will be easier to implement than !

Efficient Multirate Filter Structures

Rate-changing appears expensive computationally, since for both decimation and interpolation the lowpass filter is implemented at the higher rate. However, this is not necessary.

Interpolation

For the interpolator, most of the samples in the upsampled signal are zero, and thus require no computation. ([link])

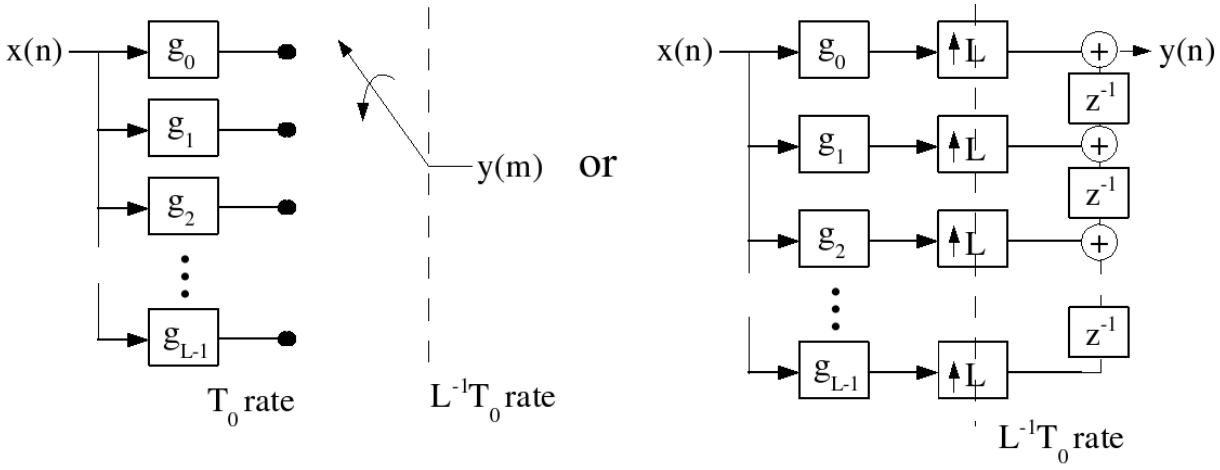


For $m = L\left\lfloor \frac{m}{L} \right\rfloor + m \bmod L$ and $p = m \bmod L$,
Equation:

$$\begin{aligned} x_1(m) &= \sum_{m}^{N_2}{}_{N_1} h_{Lp}(m) y(m) \\ &= \sum_{k}^{\frac{N_2}{L}}{}_{\frac{N_1}{L}} g_p(k) x_0\left(\left\lfloor \frac{m}{L} \right\rfloor - k\right) \end{aligned}$$

$$g_p(n) = h(Ln + p)$$

Pictorially, this can be represented as in [link].



These are called **polyphase structures**, and the $g_p(n)$ are called **polyphase filters**.

Computational cost
If $h(m)$ is a length-N filter:

- No simplification: $\frac{N}{T_1} = \frac{LN}{T_0} \frac{\text{computations}}{\text{sec}}$
- Polyphase structure: $L \frac{L}{N} \frac{1}{T_0^o} \quad \frac{\text{computations}}{\text{sec}} = \frac{N}{T_0}$ where L is the number of filters, $\frac{N}{L}$ is the taps/filter, and $\frac{1}{T_0}$ is the rate.

Thus we save a factor of L by not being dumb.

Note: For a given precision, N is proportional to L, (why?), so the computational cost does increase with the interpolation rate.

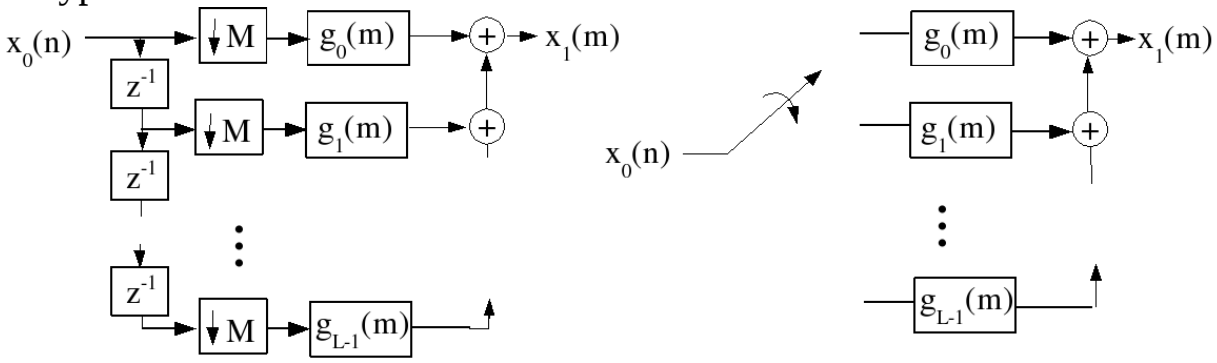
Note: Can similar computational savings be obtained with IIR structures?

Efficient Decimation Structures

We only want every Mth output, so we compute only the outputs of interest. ([link])

$$x_1(m) = \sum_{k=N_1}^{N_2} x_0(Lm-k)h(k)$$

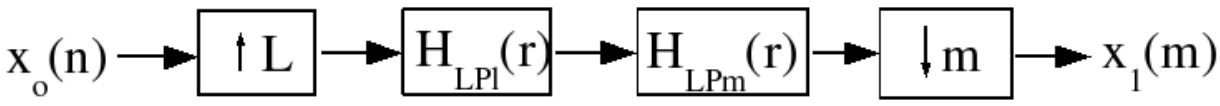
Polyphase Decimation Structure



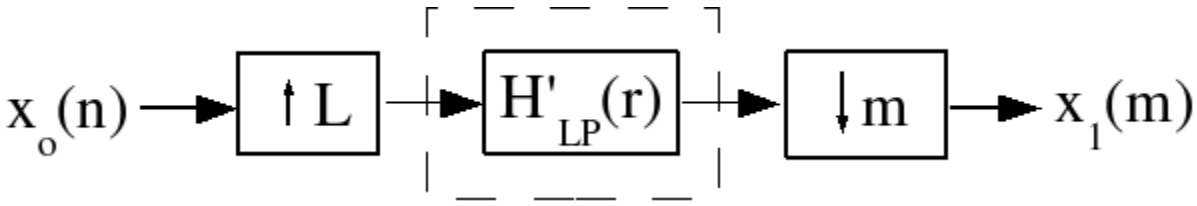
The decimation structures are flow-graph reversals of the interpolation structure. Although direct implementation of the full filter for every Mth sample is obvious and straightforward, these polyphase structures give some idea as to how one might evenly partition the computation over M cycles.

Efficient L/M rate changers

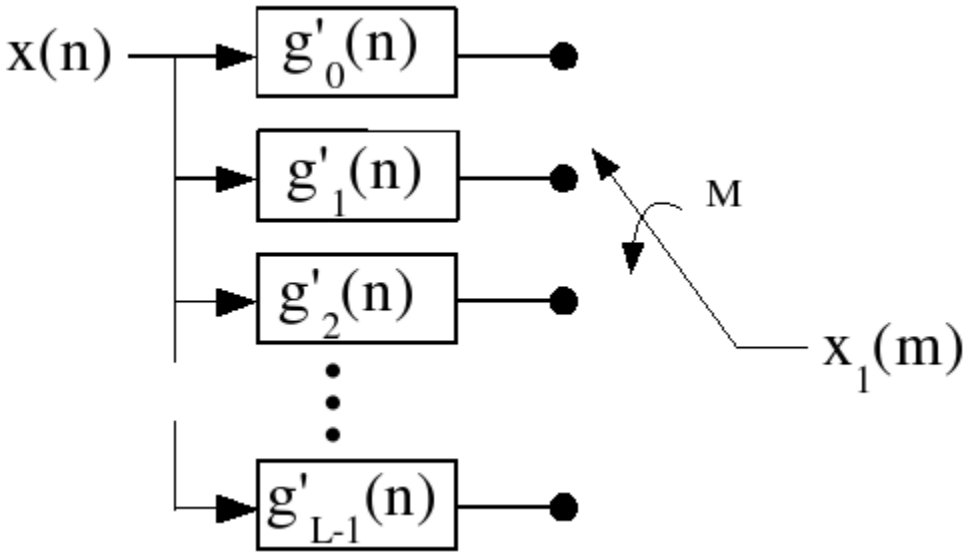
Interpolate by L and decimate by M ([link]).



Combine the lowpass filters ([link]).



We can couple the lowpass filter either to the interpolator or the decimator to implement it efficiently ([link]).



Of course we only compute the polyphase filter output selected by the decimator.

Computational Cost

Every $T_1 = \frac{M}{L} T_0$ seconds, compute one polyphase filter of length $\frac{N}{L}$, or

$$\frac{\frac{N}{L}}{T_1} = \frac{\frac{N}{L}}{\frac{M}{L} T_0} = \frac{N}{M T_0} \frac{\text{multiplies}}{\text{second}}$$

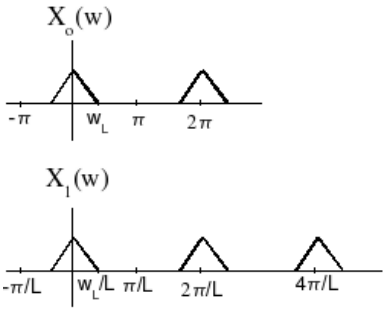
However, note that N is proportional to $\max\{L, M\}$.

Filter Design for Multirate Systems

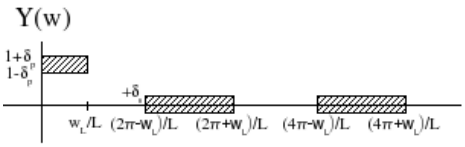
The filter design techniques learned earlier can be applied to the design of filters in multirate systems, with a few twists.

Example:
Design a factor-of-L interpolator for use in a CD player, we might wish that the out-of-band error be below the least significant bit, or 96dB down, and $< 0.05\%$ error in the passband, so these specifications could be used for optimal L_∞ filter design.

In a CD player, the sampling rate is 44.1kHz, corresponding to a Nyquist frequency of 22.05kHz, but the sampled signal is bandlimited to 20kHz. This leaves a small transition band, from 20kHz to 24.1kHz. However, note that in any case where the signal spectrum is zero over some band, this introduces **other** zero bands in the scaled, replicated spectrum ([link]).



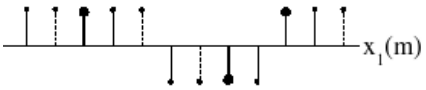
So we need only control the filter response in the stopbands over the frequency regions with nonzero energy. ([link])



The extra "don't care" bands allow a given set of specifications to be satisfied with a shorter-length filter.

Direct polyphase filter design

Note that in an integer-factor interpolator, each set of output samples $x_1(Ln+p)$, $p = \{0, 1, \ldots, L-1\}$, is created by a different polyphase filter $g_p(n)$, which has no interaction with the other polyphase filters except in that they each interpolate the same signal. We can thus treat the design of each polyphase filter **independently**, as an $\frac{N}{L}$-length filter design problem. ([link])



Each $g_p(n)$ produces samples $x_1(Ln+p) = x_0\ \ n+\frac{p}{L}\ \ $, where $n+\frac{p}{L}$ is not an integer. That is, $g_p(n)$ is to produce an output signal (at a T_0 rate) that is $x_0(n)$ time-advanced by a non-integer advance $\frac{p}{L}$.

The desired response of this polyphase filter is thus

$$H_{Dp}(\omega) = e^{\frac{i\omega p}{L}}$$

for $|\omega| \leq \pi$, an all-pass filter with a linear, non-integer, phase. Each polyphase filter can be designed independently to approximate this response according to any of the design criteria developed so far.
Exercise:

Problem: What should the polyphase filter for $p = 0$ be?

Solution:

A delta function: $h_0(n) = \delta(n')$

Example:
Least-squares Polyphase Filter Design

- **Deterministic x(n)** Minimize

$$\sum_{n=-\infty}^{\infty} \left(|x(n) - x_d(n)|\right)^2$$

  Given $x(n) = x(n)^* h(n)$ and $x_d(n) = x(n)^* h_d(n)$. Using Parseval's theorem, this becomes
  Equation:

$$\begin{aligned} \min \quad \sum_{n=-\infty}^{\infty} \left(|x(n) - x_d(n)|\right)^2 &= \min \quad \frac{1}{2\pi} \int_{-\pi}^{\pi} \left(|X(\omega)H(\omega) - X(\omega)H_d(\omega)|\right)^2 \,\mathrm{d}\,\omega \\ &= \min \quad \frac{1}{2\pi} \int_{-\pi}^{\pi} |H(\omega) - H_d(\omega)| \left(|X(\omega)|\right)^2 \,\mathrm{d}\,\omega \end{aligned}$$

  This is simply weighted least squares design, with $\left(|X(\omega)|\right)^2$ as the weighting function.
- **stochastic X(ω)**
  Equation:

$$\begin{aligned} \min \quad E\ \left(|x(n) - x_d(n)|\right)^2 &= E\ \left(|x(n)^*(h(n) - h_d(n))|\right)^2 \\ &= \min \quad \frac{1}{2\pi} \int_{-\pi}^{\pi} \left(|H_d(\omega) - H(\omega)|\right)^2 S_{xx}(\omega) \,\mathrm{d}\,\omega \end{aligned}$$

$S_{xx}(\omega)$ is the power spectral density of x.

$$S_{xx}(\omega) = \mathrm{DTFT}\left[r_{xx}(k)\right]$$

$$r_{xx}(k) = E\ x(k+l)x(l)$$

Again, a weighted least squares filter design problem.

Exercise:

Problem: Is it feasible to use IIR polyphase filters?

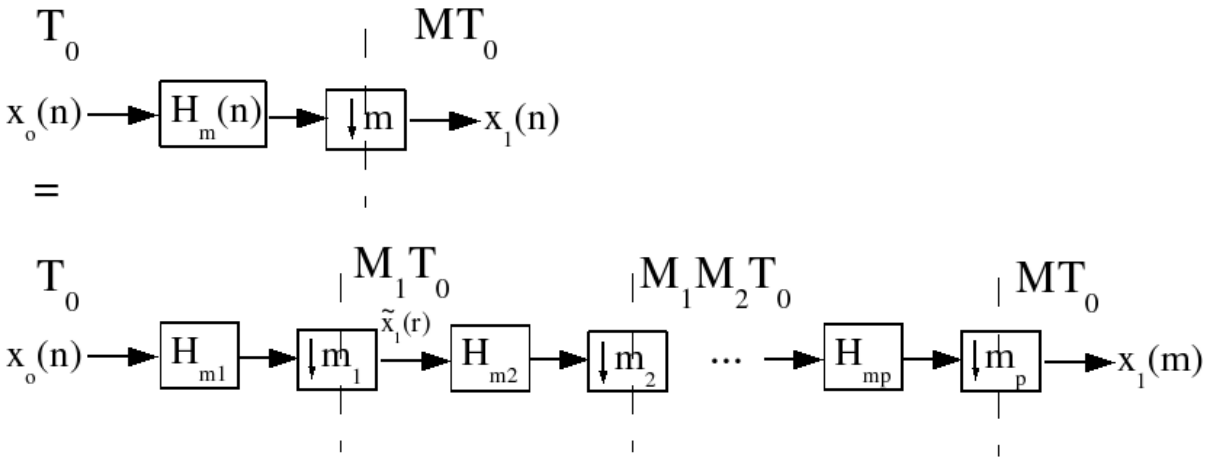
Solution:

The recursive feedback of previous outputs means that portions of each IIR polyphase filter must be computed for every input sample; this usually makes IIR filters more expensive than FIR implementations.

Multistage Multirate Systems

Multistage multirate systems are often more efficient. Suppose one wishes to decimate a signal by an integer factor M, where M is a composite integer $M = M_1 M_2 \ldots M_p = \prod_{i=1}^{p} M_i$. A decimator can be implemented in a multistage fashion by first decimating by a factor M_1 , then decimating this signal by a factor M_2 , etc. ([link])

Multistage decimator



Multistage implementations are of significant practical interest only if they offer significant computational savings. In fact, they often do!

The computational cost of a **single-stage** interpolator is:

$$\frac{N}{MT_0}\frac{\text{taps}}{\text{sec}}$$

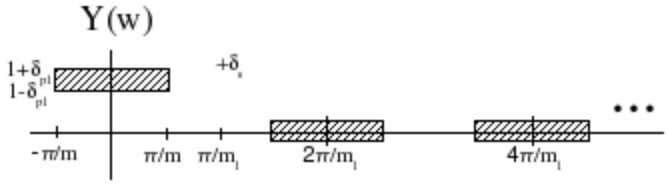
The computational cost of a **multistage** interpolator is:

$$\frac{N_1}{M_1 T_0} + \frac{N_2}{M_1 M_2 T_0} + \ldots + \frac{N_p}{MT_0}$$

The first term is the most significant, since the rate is highest. Since $N_i \propto M_i$ for a lowpass filter, it is not immediately clear that a multistage system should require less computation. However, the multistage structure relaxes the requirements on the filters, which reduces their length and makes the overall computation less.

Filter design for Multi-stage Structures

Ostensibly, the first-stage filter must be a lowpass filter with a cutoff at $\frac{\pi}{M_1}$, to prevent aliasing after the downsampler. However, note that aliasing outside the **final overall** passband $|\omega| < \frac{\pi}{M}$ is of no concern, since it will be removed by later stages. We only need prevent aliasing into the band $|\omega| < \frac{\pi}{M}$; thus we need only specify the passband over the interval $|\omega| < \frac{\pi}{M}$, and the stopband over the intervals $\omega \in \left[\frac{2\pi k}{M_1} - \frac{\pi}{M}, \frac{2\pi k}{M_1} + \frac{\pi}{M}\right]$, for $k \in \{1, \ldots, M-1\}$. ([link])



Of course, we don't want **gain** in the transition bands, since this would need to be suppressed later, but otherwise we don't care about the response in those regions. Since the transition bands are so large, the required filter turns out to be quite short. The final stage has no "don't care" regions; however, it is operating at a low rate, so it is relatively unimportant if the final filter turns out to be rather long!

L-infinity Tolerances on the Pass and Stopbands

The overall response is a cascade of multiple filters, so the worst-case overall passband deviation, assuming all the peaks just happen to line up, is

$$1 + \delta_{p_{\text{ov}}} = \prod_{i=1}^{p} 1 + \delta_{p_i}$$

$$1 - \delta_{p_{\text{ov}}} = \prod_{i=1}^{p} 1 - \delta_{p_i}$$

So one could choose all filters to have equal specifications and require for each-stage filter. For $\delta_{p_{\text{ov}}} \ll 1$,

$$1 + \delta_{p_i}^{+} \leq \sqrt[p]{1 + \delta_{p_{\text{ov}}}} \simeq 1 + p^{-1}\delta_{p_{\text{ov}}}$$

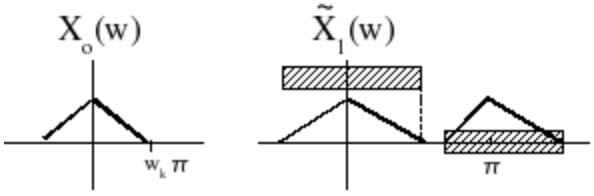
$$1 - \delta_{p_i}^{-} \leq \sqrt[p]{1 - \delta_{p_{\text{ov}}}} \simeq 1 - p^{-1}\delta_{p_{\text{ov}}}$$

Alternatively, one can design later stages (at lower computation rates) to compensate for the passband ripple in earlier stages to achieve exceptionally accurate passband response.

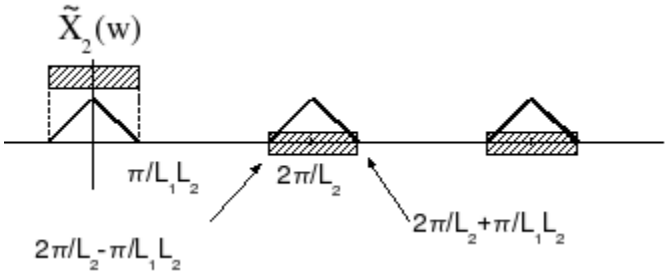
δ_s remains essentially unchanged, since the worst-case scenario is for the error to alias into the passband and undergo no further suppression in subsequent stages.

Interpolation

Interpolation is the flow-graph reversal of the multi-stage decimator. The first stage has a cutoff at $\frac{\pi}{L}$ ([link]):



However, all subsequent stages have large bands without signal energy, due to the earlier stages ([link]):

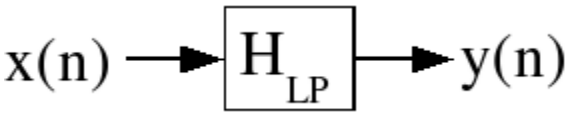


The order of the filters is reversed, but otherwise the filters are identical to the decimator filters.

Efficient Narrowband Lowpass Filtering

A very narrow lowpass filter requires a very long FIR filter to achieve reasonable resolution in the frequency response. However, were the input sampled at a lower rate, the cutoff frequency would be correspondingly higher, and the filter could be shorter!

The transition band is also broader, which helps as well. Thus, [link] can be implemented as [link].

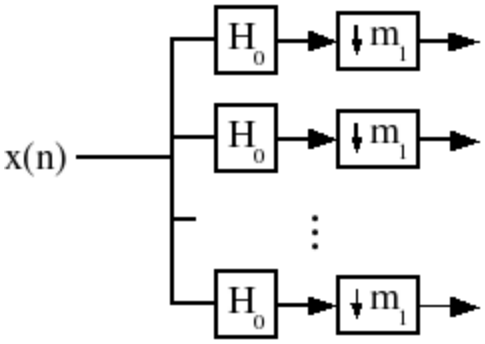




and in practice the inner lowpass filter can be coupled to the decimator or interpolator filters. If the decimator and interpolator are implemented as multistage structures, the overall algorithm can be dramatically more efficient than direct implementation!

DFT-Based Filterbanks

One common application of multirate processing arises in multirate, multi-channel filter banks ([link]).



One application is separating frequency-division-multiplexed channels. If the filters are narrowband, the output channels can be decimated without significant aliasing.

Such structures are especially attractive when they can be implemented efficiently. For example, if the filters are simply frequency modulated (by $e^{-i\frac{2\pi k}{L}n}$) versions of each other, they can be efficiently implemented using FFTs!

Furthermore, there are classes of filters called **perfect reconstruction filters** which are of finite length but from which the signal can be reconstructed exactly (using all M channels), even though the output of each channel experiences aliasing in the decimation step. These types of filterbanks have received a lot of research attention, culminating in wavelet theory and techniques.

Uniform DFT Filter Banks

Suppose we wish to split a digital input signal into N frequency bands, uniformly spaced at center frequencies $\omega_k = \frac{2\pi k}{N}$, for $0 \leq k \leq N-1$. Consider also a lowpass filter $h(n)$, $H(\omega) \simeq \begin{matrix} 1 & \text{if} & |\omega| < \frac{\pi}{N} \\ 0 & \text{otherwise} \end{matrix}$. Bandpass filters can be constructed which have the frequency response

$$H_k(\omega) = H \quad \omega + \frac{2\pi k}{N}$$

from

$$h_k(n) = h(n)e^{- \, i\frac{2\pi kn}{N}}$$

The output of the *k*th bandpass filter is simply (assume $h(n)$ are FIR)
Equation:

$$\begin{aligned} x(n)^* h_k(n) &= \sum_{m \;\; 0}^{M-1} x(n-m)h(m)e^{- \, i\frac{2\pi km}{N}} \\ &= y_k(n) \end{aligned}$$

This looks suspiciously like a DFT, except that $M \neq N$, in general. However, if we fix $M = N$, then we can compute **all** $y_k(n)$ outputs simultaneously using an FFT of $x(n-m)h(m)$: The
$\text{kth FFT frequency output} = y_k(n)$! So the cost of computing all of these filter banks outputs is $O[N \log N]$, rather than N^2, per a given n. This is very useful for efficient implementation of **transmultiplexors** (FDM to TDM).
Exercise:

Problem:

How would we implement this efficiently if we wanted to decimate the individual channels $y_k(n)$ by a factor of N, to their approximate Nyquist bandwidth?

Solution:

Simply step by N time samples between FFTs.

Exercise:

Problem:

Do you expect significant aliasing? If so, how do you propose to combat it? Efficiently?

Solution:

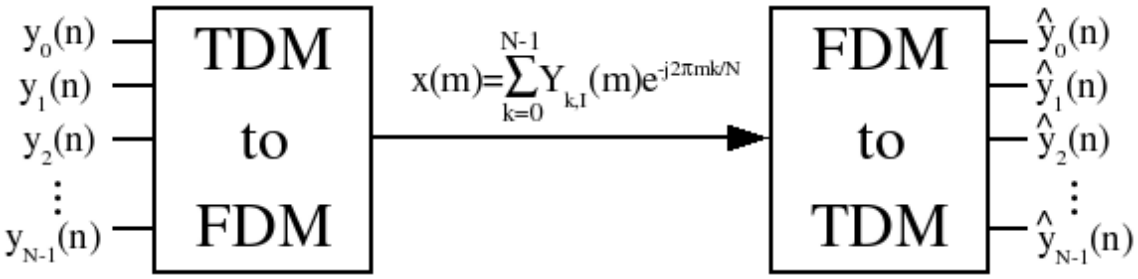
Aliasing should be expected. There are two ways to reduce it:

1. Decimate by less ("oversample" the individual channels) such as decimating by a factor of $\frac{N}{2}$. This is efficiently done by time-stepping by the appropriate factor.
2. Design better (and thus longer) filters, say of length LN. These can be efficiently computed by producing only N (every Lth) FFT outputs using simplified FFTs.

Exercise:

Problem:

How might one convert from N input channels into an FDM signal efficiently? ([link])



Note: Such systems are used throughout the telephone system, satellite communication links, etc.

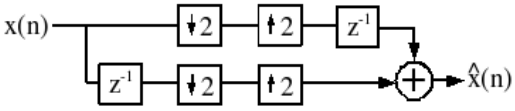
Solution:

Use an FFT and an inverse FFT for the modulation (TDM to FDM) and demodulation (FDM to TDM), respectively.

Quadrature Mirror Filterbanks (QMF)

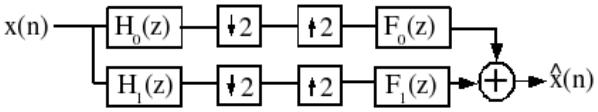
Although the DFT filterbanks are widely used, there is a problem with aliasing in the decimated channels. At first glance, one might think that this is an insurmountable problem and must simply be accepted. Clearly, with FIR filters and maximal decimation, aliasing will occur. However, a simple example will show that it is possible to **exactly cancel out aliasing** under certain conditions!!!

Consider the following trivial filterbank system, with two channels. ([link])



Note $\widehat{x}(n) = x(n)$ with no error whatsoever, although clearly aliasing occurs in both channels! Note that the overall data rate is still the Nyquist rate, so there are clearly enough degrees of freedom available to reconstruct the data, if the filterbank is designed carefully. However, this isn't splitting the data into separate frequency bands, so one questions whether something other than this trivial example could work.

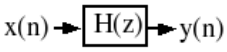
Let's consider a general two-channel filterbank, and try to determine conditions under which aliasing can be cancelled, and the signal can be reconstructed perfectly ([link]).



Let's derive $\widehat{x}(n)$, using z-transforms, in terms of the components of this system. Recall ([link]) is equivalent to

$$Y(z) = H(z)X(z)$$

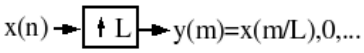
$$Y(\omega) = H(\omega)X(\omega)$$



and note that ([link]) is equivalent to

$$Y(z) = \sum_{m=-\infty}^{\infty} x(m)z^{-(Lm)} = x\left(z^L\right)$$

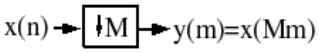
$$Y(\omega) = X(L\omega)$$



and ([link]) is equivalent to

$$Y(z) = \frac{1}{M}\sum_{k=0}^{M-1} X\left(z^{\frac{1}{M}}W_M^k\right)$$

$$Y(\omega) = \frac{1}{M}\sum_{k=0}^{M-1} X\left(\frac{\omega}{M} + \frac{2\pi k}{M}\right)$$



$Y(z)$ is derived in the downsampler as follows:

$$Y(z) = \sum_{m=-\infty}^{\infty} x(Mm) z^{-m}$$

Let $n = Mm$ and $m = \frac{n}{M}$, then

$$Y(z) = \sum_{n=-\infty}^{\infty} x(n) \sum_{p=-\infty}^{\infty} \delta(n - Mp) z^{-\frac{n}{M}}$$

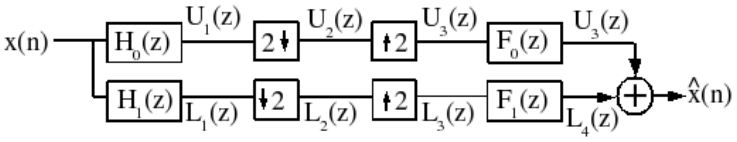
Now
Equation:

$$\begin{aligned} x(n) \textstyle\sum_{p=-\infty}^{\infty} \delta(n - Mp) &= \mathrm{IDFT}\left[x(\omega) * \frac{2\pi}{M} \sum_{k=0}^{M-1} \delta\left(\omega - \frac{2\pi k}{M}\right)\right] \\ &= \mathrm{IDFT}\left[\frac{2\pi}{M} \sum_{k=0}^{M-1} X\left(\omega - \frac{2\pi k}{M}\right)\right] \\ &= \frac{1}{M} \sum_{k=0}^{M-1} X(n) W_M^{-nk}\Big|_{W_M = e^{-\frac{i2\pi}{M}}} \end{aligned}$$

so
Equation:

$$\begin{aligned} Y(z) &= \textstyle\sum_{n=-\infty}^{\infty} \left(\frac{1}{M} \sum_{k=0}^{M-1} x(n) W_M^{-nk}\right) z^{-\frac{n}{M}} \\ &= \frac{1}{M} \sum_{k=0}^{M-1} x(n) \left(W_M^{+k} z^{\frac{1}{M}}\right)^{-n} \\ &= \frac{1}{M} \sum_{k=0}^{M-1} X\left(z^{\frac{1}{M}} W_M^{k}\right) \end{aligned}$$

Armed with these results, let's determine $\widehat{X}(z) \Leftrightarrow \widehat{x}(n)$. ([link])



Note

$$U_1(z) = X(z) H_0(z)$$

$$U_2(z) = \frac{1}{2} \sum_{k=0}^{1} X\left(z^{\frac{1}{2}} e^{-\frac{i2\pi k}{2}}\right) H_0\left(z^{\frac{1}{2}} e^{-(i\pi k)}\right) = \frac{1}{2} X\left(z^{\frac{1}{2}}\right) H_0\left(z^{\frac{1}{2}}\right) + \frac{1}{2} X\left(-z^{\frac{1}{2}}\right) H_0\left(-z^{\frac{1}{2}}\right)$$

$$U_3(z) = \frac{1}{2} X(z) H_0(z) + \frac{1}{2} X(-z) H_0(-z)$$

$$U_4(z) = \frac{1}{2} F_0(z) H_0(z) X(z) + \frac{1}{2} F_0(z) H_0(-z) X(-z)$$

and

$$L_4(z) = \frac{1}{2}F_1(z)H_1(z)X(z) + \frac{1}{2}F_1(z)H_1(-z)X(-z) = \frac{1}{2}F_1(z)H_1(z)X(z) + \frac{1}{2}F_1(z)H_1(-z)X(-z)$$

Finally then,
Equation:

$$\begin{aligned}\widehat{X}(z) &= U_4(z) + L_4(z) \\ &= \tfrac{1}{2}\left(H_0(z)F_0(z)X(z) + H_0(-z)F_0(z)X(-z) + H_1(z)F_1(z)X(z) + H_1(-z)F_1(z)X(-z)\right) \\ &= \tfrac{1}{2}\left(H_0(z)F_0(z) + H_1(z)F_1(z)\right)X(z) + \tfrac{1}{2}\left(H_0(-z)F_0(z) + H_1(-z)F_1(z)\right)X(-z)\end{aligned}$$

Note that the $X(-z) \rightarrow X(\omega + \pi)$ corresponds to the aliasing terms!

There are four things we would like to have:

1. No aliasing distortion
2. No phase distortion (overall linear phase → simple time delay)
3. No amplitude distortion
4. FIR filters

No aliasing distortion
By insisting that $H_0(-z)F_0(z) + H_1(-z)F_1(z) = 0$, the $X(-z)$ component of $\widehat{X}(z)$ can be removed, and all aliasing will be eliminated! There may be many choices for H_0, H_1, F_0, F_1 that eliminate aliasing, but most research has focused on the choice

$$F_0(z) = H_1(-z) : F_1(z) = -H_0(-z)$$

We will consider only this choice in the following discussion.

Phase distortion
The transfer function of the filter bank, with aliasing cancelled, becomes
$T(z) = \frac{1}{2}\left(H_0(z)F_0(z) + H_1(z)F_1(z)\right)$, which with the above choice becomes
$T(z) = \frac{1}{2}\left(H_0(z)H_1(-z) - H_1(z)H_0(-z)\right)$. We would like $T(z)$ to correspond to a linear-phase filter to eliminate phase distortion: Call

$$P(z) = H_0(z)H_1(-z)$$

Note that

$$T(z) = \frac{1}{2}\left(P(z) - P(-z)\right)$$

Note that $P(-z) \Leftrightarrow (-1)^n p(n)$, and that if $p(n)$ is a linear-phase filter, $(-1)^n p(n)$ is also (perhaps of the opposite symmetry). Also note that the sum of two linear-phase filters of the same symmetry (i.e., length of $p(n)$ must be **odd**) is also linear phase, so if $p(n)$ is an odd-length linear-phase filter, there will be no phase distortion. Also note that

$$Z^{-1}(p(z) - p(-z)) = p(n) - (-1)^n p(n) = \begin{cases} 2p(n) & \text{if n is odd} \\ 0 & \text{if n is even} \end{cases}$$

means $p(n) = 0$, when n is even. If we choose $h_0(n)$ and $h_1(n)$ to be linear phase, $p(n)$ will also be linear phase. Thus by choosing $h_0(n)$ and $h_1(n)$ to be FIR linear phase, we eliminate phase distortion and get FIR filters as well (condition 4).

Amplitude distortion
Assuming aliasing cancellation and elimination of phase distortion, we might also desire no amplitude distortion ($|T(\omega)| = 1$). All of these conditions require

$$T(z) = \frac{1}{2}\left(H_0(z)H_1(-z) - H_1(z)H_0(-z)\right) = cz^{-D}$$

where c is some constant and D is a linear phase delay. $c = 1$ for $|T(\omega)| = 1$. It can be shown by considering that the following can be satisfied!

$$T(z) = P(z) - P(-z) = 2cz^{-D} \Leftrightarrow \begin{cases} 2p(z) = 2c\delta(n-D) \ \text{ if } \ \text{n is odd} \\ p(n) = \text{anything } \ \text{ if } \ \text{n is even} \end{cases}$$

Thus we require

$$P(z) = \sum_{n=0}^{N'} p(2n)z^{-(2n)} + z^{-D}$$

Any factorization of a $P(z)$ of this form, $P(z) = A(z)B(z)$ can lead to a Perfect Reconstruction filter bank of the form

$$H_0(z) = A(z)$$

$$H_1(-z) = B(z)$$

[This result is attributed to Vetterli.] A well-known special case (Smith and Barnwell)

$$H_1(z) = -\left(z^{-(2D)+1}H_0\left(-z^{-1}\right)\right)$$

Design techniques exist for optimally choosing the coefficients of these filters, under all of these constraints.
Equation:

Quadrature Mirror Filters

$$H_1(z) = H_0(-z) \Leftrightarrow H_1(\omega) = H_0(\pi + \omega) = H_0^*(\pi - \omega)$$

for real-valued filters. The frequency response is "mirrored" around $\omega = \frac{\pi}{2}$. This choice leads to $T(z) = {H_0}^2(z) - {H_0}^2(-z)$: it can be shown that this can be a perfect reconstruction system only if

$$H_0(z) = c_0 z^{-(2n_0)} + c_1 z^{-(2n_1)}$$

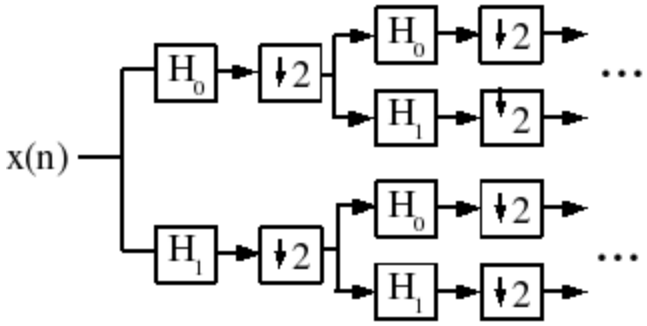
which isn't a very flexible choice of filters, and not a very good lowpass! The Smith and Barnwell approach is more commonly used today.

M-Channel Filter Banks

The theory of M-band QMFBs and PRFBs has been investigated recently. Some results are available.

Tree-structured filter banks

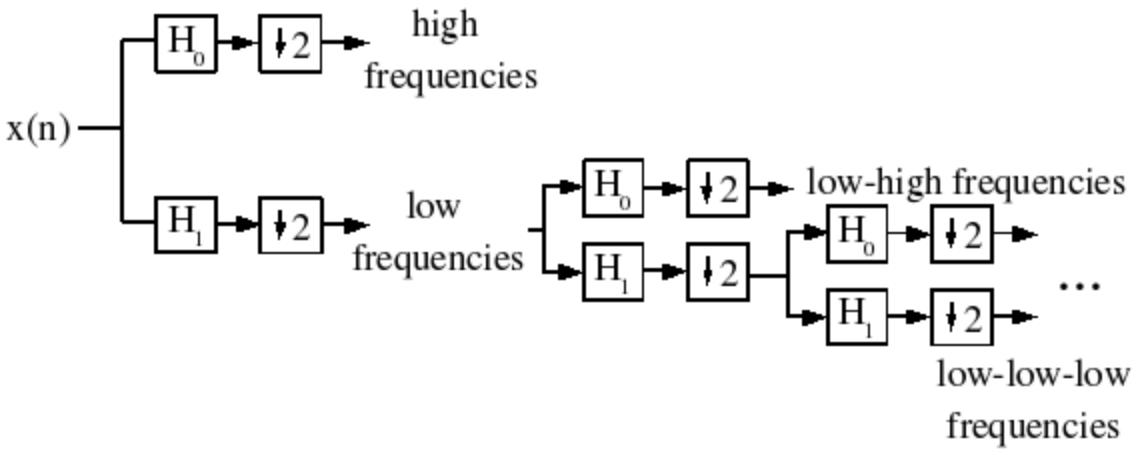
Once we have a two-band PRFB, we can continue to split the subbands with similar systems! ([link])



Thus we can recursively decompose a signal into 2 bands, each sampled at 2 th the rate of the original signal, and reconstruct exactly! Due to the tree structure, this can be quite efficient, and in fact close to the efficiency of an FFT filter bank, which does **not** have perfect reconstruction.

Wavelet decomposition

We need not split both the upper-frequency and lower-frequency bands identically. ([link])



This is good for image coding, because the energy tends to be distributed such that after a wavelet decomposition, each band has roughly equal energy.